VISIT…

The Linux TCP/IP Stack: Networking for Embedded Systems

by Thomas F. Herbert

Charles River Media © 2004 (600 pages)

ISBN:1584502843

Written for embedded systems programmers and engineers, as well as networking professionals, this in-depth guide provides an inside look at the entire process of implementing and using the Linux TCP/IP stack in embedded systems projects.

Table of Contents

The Linux TCP/IP Stack—Networking for Embedded Systems

Chapter 1 - Introduction

Chapter 2 - Broadband Networking Protocols of Yesterday and Today

Chapter 3 - TCP/IP in Embedded Systems

Chapter 4 - Linux Networking Interfaces and Device Drivers

Chapter 5 - Linux Sockets

Chapter 6 - The Linux TCP/IP Stack

Chapter 7 - Socket Buffers and Linux Memory Allocation

Chapter 8 - Sending the Data from the Socket through UDP and TCP

Chapter 9 - The Network Layer, IP

Chapter 10 - Receiving Data in the Transport Layer, UDP, and TCP

Chapter 11 - Internet Protocol Version 6 (IPv6)

Appendix A - RFCs

Appendix B - About the CD-ROM

Bibliography

Index

List of Figures

List of Tables

The Linux TCP/IP Stack—Networking for Embedded Systems

Thomas F. Herbert

Copyright 2004 by CHARLES RIVER MEDIA, INC.
All rights reserved.

No part of this publication may be reproduced in any way, stored in a retrieval system of any type, or transmitted by any means or media, electronic or mechanical, including, but not limited to, photocopy, recording, or scanning, without prior permission in writing from the publisher.

Acquisitions Editor
James Walsh

Cover Design
The Printed Image

CHARLES RIVER MEDIA, INC.
10 Downer Avenue
Hingham, Massachusetts 02043
781-740-0400
781-740-8816 (FAX)
info@charlesriver.com
www.charlesriver.com

Thomas Herbert. The Linux TCP/IP Stack: Networking for Embedded Systems.
ISBN: 1-58450-284-3

All brand names and product names mentioned in this book are trademarks or service marks of their respective companies. Any omission or misuse (of any kind) of service marks or trademarks should not be regarded as intent to infringe on the property of others. The publisher recognizes and respects all marks used by companies, manufacturers, and developers as a means to distinguish their products.

Library of Congress Cataloging-in-Publication Data
Herbert, Thomas (Thomas F.)
The Linux TCP/IP stack : networking for embedded systems / Thomas Herbert.— 1st ed.
p. cm.
ISBN 1-58450-284-3 (pbk. with cd-rom : alk. paper)
1. Linux. 2. Operating systems (Computers) 3. TCP/IP (Computer network protocol) A04.
Embedded computer systems. I. Title.

QA76.73.O63H47 2004
005.4’32—dc22
2004005014

Printed in the United States of America
04 7 6 5 4 3 2 First Edition

CHARLES RIVER MEDIA titles are available for site license or bulk purchase by institutions, user groups, corporations, etc. For additional information, please contact the Special Sales Department at 781-740-0400.

Requests for replacement of a defective CD-ROM must be accompanied by the original disc, your mailing address, telephone number, date of purchase and purchase price. Please state the nature of the problem, and send the information to CHARLES RIVER MEDIA, INC., 10 Downer Avenue, Hingham, Massachusetts 02043. CRM’s sole obligation to the purchaser is to replace the disc, based on defective materials or faulty workmanship, but not on the operation or functionality of the product.

LIMITED WARRANTY AND DISCLAIMER OF LIABILITY

THE CD-ROM THAT ACCOMPANIES THE BOOK MAY BE USED ON A SINGLE PC ONLY. THE LICENSE DOES NOT PERMIT THE USE ON A NETWORK (OF ANY KIND). YOU FURTHER AGREE THAT THIS LICENSE GRANTS PERMISSION TO USE THE PRODUCTS CONTAINED HEREIN, BUT DOES NOT GIVE YOU RIGHT OF OWNERSHIP TO ANY OF THE CONTENT OR PRODUCT CONTAINED ON THIS CD-ROM. USE OF THIRD-PARTY SOFTWARE CONTAINED ON THIS CD-ROM IS LIMITED TO AND SUBJECT TO LICENSING TERMS FOR THE RESPECTIVE PRODUCTS.

CHARLES RIVER MEDIA, INC. (“CRM”) AND/OR ANYONE WHO HAS BEEN INVOLVED IN THE WRITING, CREATION, OR PRODUCTION OF THE ACCOMPANYING CODE (“THE SOFTWARE”) OR THE THIRD-PARTY PRODUCTS CONTAINED ON THE CD-ROM OR TEXTUAL MATERIAL IN THE BOOK, CANNOT AND DO NOT WARRANT THE PERFORMANCE OR RESULTS THAT MAY BE OBTAINED BY USING THE SOFTWARE OR CONTENTS OF THE BOOK. THE AUTHOR AND PUBLISHER HAVE USED THEIR BEST EFFORTS TO ENSURE THE ACCURACY AND FUNCTIONALITY OF THE TEXTUAL MATERIAL AND PROGRAMS CONTAINED HEREIN. WE HOWEVER, MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, REGARDING THE PERFORMANCE OF THESE PROGRAMS OR CONTENTS. THE SOFTWARE IS SOLD “AS IS” WITHOUT WARRANTY (EXCEPT FOR DEFECTIVE MATERIALS USED IN MANUFACTURING THE DISK OR DUE TO FAULTY WORKMANSHIP).

THE AUTHOR, THE PUBLISHER, DEVELOPERS OF THIRD-PARTY SOFTWARE, AND ANYONE INVOLVED IN THE PRODUCTION AND MANUFACTURING OF THIS WORK SHALL NOT BE LIABLE FOR DAMAGES OF ANY KIND ARISING OUT OF THE USE OF

(OR THE INABILITY TO USE) THE PROGRAMS, SOURCE CODE, OR TEXTUAL MATERIAL CONTAINED IN THIS PUBLICATION. THIS INCLUDES, BUT IS NOT LIMITED TO, LOSS OF REVENUE OR PROFIT, OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THE PRODUCT.

THE SOLE REMEDY IN THE EVENT OF A CLAIM OF ANY KIND IS EXPRESSLY LIMITED TO REPLACEMENT OF THE BOOK AND/OR CD-ROM, AND ONLY AT THE DISCRETION OF CRM.

THE USE OF "IMPLIED WARRANTY" AND CERTAIN "EXCLUSIONS" VARIES FROM STATE TO STATE, AND MAY NOT APPLY TO THE PURCHASER OF THIS PRODUCT.

Acknowledgments

While working on this book, I often thought of this endeavor as a lonely and particularly solitary effort in attempting to unravel this complex and intricate body of software. In fact, though, I have not been alone. Many influenced me to take on this project, and there are many without whom I would not have been able to complete this task. I owe enormous gratitude to the many people who have helped me along the path that led to this book, some of whom are mentioned here.

First and foremost, I am indebted to my late parents, George and Dorothy Herbert. My father is my main inspiration. He was a self-taught and very intelligent man with a strong life-long interest in learning, literature, and writing. He brought me up in a house filled with books, and at a very young age, I came to see the value of the printed word. Both my father and my mother always told me that I could do anything and accomplish anything I put my mind to. Whenever I stumble along the way, I remember my mother's advice that with hard work, courage, and creativity I can achieve any goal in which I believe.

I feel some gratitude to the embedded development community, which has been my profession and source of income for most of the last 20 years. There have been a few key people in my business and professional life who guided my path leading to my publishing activities and ultimately to the writing of this book. I first thought about publishing articles about networking in a commercial magazine after talking with Marty Leisner. Marty was a coworker of mine at Xerox in the mid-1990s and an early proponent of the open-source philosophy. In addition, at that time, I polished my writing skills when my manager, Hugo Buitano, would often ask for my assistance to produce "hand crafted" e-mails when a complex and sensitive subject would come up and needed handling accurately but with tact and diplomacy. I should also mention Lindsey Vereen, editor of "Embedded Systems Programming," who noticed my paper in the conference proceedings and said that it was "well written for a conference paper."

I have enormous respect and am extremely grateful to all of the people involved in the open-source community and the Linux development project. The open-source movement grows in strength all the time and has defined a new way of doing business in our industry. It is opening markets around the world and proving to be an effective counterweight to the monopolistic

practices of some major players in our industry. I am indebted to pioneers in the open-source movement such as Richard Stallman, who begin the Free Software Foundation and first used the term *free software*, meaning freedom to create, modify, and improve. I am particularly indebted to Linux Torvalds, the original developer of the Linux kernel.

In addition, there are all the people involved in the long development and improvement of TCP/IP, beginning with the late Jon Postel who was the editor of the RFCs for many years. He authored many of the early RFCs specifying how the protocols written about in this book actually work. For more information about Jon Postel, go to the Postel Center at the University of Southern California, *www.postel.org/postel.html*. I also am grateful to Gary R. Write and W. Richard Stevens, authors of what is clearly the definitive work about the internal functioning of any TCP/IP implementation. In this book, I hope I am at least coming close to living up to the high standard they have created. Finally, I extend my thanks to the contributors to the Linux TCP/IP implementation. There are too many to mention here, but I must name one person in particular, Alexey Kuznetsov, who has contributed a huge volume of well-documented high-quality professional code to Linux TCP/IP.

I have benefited by the association with the highly professional staff at Charles River Media, which started when David Pallai, the president and founder of Charles River Media, first approached me and invited my book proposal. James Walsh, the acquisitions editor, has been with me during the entire process. Jim has been very patient with me as I faced unanticipated problems delaying the work and needed extra time to dig out the bits or follow all the threads through complexities in the code. He has supported me through changes in goals for this project with the addition of IPv6 and the 2.6 kernel. Bryan Davidson, the production coordinator, has been extremely helpful. He took the extra time to answer my arcane formatting questions and showed flexibility as we moved this work through production.

I want to acknowledge the work of my reviewers, Jim Lieb and David McLain. Jim's review of the first part of the book helped me to focus on the practical needs of my readers and the market. I want to extend special thanks to David, who went way beyond the call of duty. His knowledge of the Linux kernel code, computer networking, memory allocation methods, and good software engineering practice was very helpful. He carefully checked all my explanations against the Linux source code and was not shy about pointing out inaccuracies, mistakes, and places where the text needed clarification.

Finally, there is my family, particularly my 14-year-old son, Aaron, and my wife of many years, Carol. Both of them have suffered my absences during the creation of this book. They have been very tolerant and patient and I apologize to them for the many hours that I spent hunched over this laptop ignoring their needs. They have been patient with me while tasks went undone and remained so when basketball games and other events were missed while I struggled to meet my deadlines. Carol never stopped believing in my ability to complete this task throughout this 14-month project. Carol is an inspiration to me in so many ways; even when the amount of work seemed overwhelming, she never lost faith in me.

Chapter 1: Introduction

Overview

This book is about the Linux implementation of TCP/IP. Certainly, some of you are eager to dive headfirst into the Linux TCP/IP source code. These readers can go straight to the later chapters beginning with Chapter 4, "Linux Networking Interfaces and Device Drivers," and proceed from there. It is difficult, however, to discuss any networking topic, including TCP/IP, without first building a foundation or a common basis of discussion. Later in the book, we dive deep into the details of the TCP/IP implementation in Linux. We will show how TCP/IP is very complex but amazingly robust, particularly as a mature and stable implementation such as Linux. It is also a versatile implementation that is suitable for desktop host systems, high-volume servers, and even routers.

In this book, we will cover some computer networking theory, but primarily, we assume a general background in data communications. The early chapters of the book provide background in data communications to show how modern implementations of TCP/IP, such as Linux, evolved as the result of changes over many years. Overall, our approach to covering the Linux TCP/IP is practical. Particularly in the later chapters, the discussion is from a developer's point of view. It is intended to show how Linux TCP/IP is at the apex of the evolution of data communications. As data communication standards and theory change, because of the open source movement, the popularity and flexibility of the Linux operating system make it one of the main platforms to implement data communications. It will be shown how the strength of TCP/IP is in its age and how it has evolved to meet modern needs. TCP/IP has been around for more than 20 years, and the Linux implementation is about 10 years old.

The general approach taken in this book is to follow the data as it flows through the networking protocols. Generally, the book is organized to follow a packet as it is sent from a networking application running in a Linux host as it flows through the stack and out the wire. Then it follows the data packet as it is received by another system until it is posted to the application code. Along the way, it shows how packets are routed, how TCP states are maintained, and how the socket interface works.

In this chapter, we will show how the Linux sources are organized. Then, we will launch the complex topic of TCP/IP by providing background material in data communication and computer networking protocols. We will talk about the Linux TCP/IP stack in general terms and will present a general background in data communication. Chapter 2, "Broadband Networking Protocols of Yesterday and Today," follows by going into data communication in more detail. Some significant networking protocols will be explored in a historical context. Although the protocols in Chapter 2 are not directly related to TCP/IP, the discussion will help to build the background needed for the detailed information in later chapters. Chapter 3, "TCP/IP in Embedded Systems," discusses TCP/IP from the standpoint of the embedded systems engineer. After some general background on network interfaces, Chapter 4 concentrates on Linux network interface drivers and how to interface them to the Linux TCP/IP stack. The socket library is the primary means for application programs to send and receive data using the TCP/IP protocols and Chapter 5, "Linux Sockets," covers sockets, including how they are implemented in Linux and

how applications exchange data with protocols in the kernel using the socket API. Chapter 6, "The Linux TCP/IP Stack," covers basic elements of the Linux TCP/IP stack, including the glue that holds the stack together. Information about general kernel facilities is included if it is used by TCP/IP. Chapter 7, "Socket Buffers and Linux Memory Allocation," includes some background on memory allocation for network systems, the Linux slab cache memory allocation system, and the specific slab caches for networking data and protocols. In addition, Chapter 7 explains socket buffers, which are the basic data structures holding packets as they flow through the networking protocols. Chapter 8, "Sending the Data from the Socket through UDP and TCP," covers the transport protocols, TCP and UDP. It follows the flow of data packets through the transport layer. Chapter 9, "The Network Layer, IP," is about IP, the network layer protocol. It covers how IP receives a packet from TCP or UDP and hands it off to the device drivers. In addition, it shows how IP receives a packet from the device drivers and queues it to the transport protocols. It includes information about the implementation of the routing tables and how routing decisions are made. Chapter 10, "Receiving Data in the Transport Layer, UDP and TCP," covers the receive side of UDP and TCP. It follows a packet as it is received from IP and is queued up to the socket for reading by the application code. Chapter 11, "Internet Protocol Version 6 (IPv6)," covers IPv6. It examines which facilities IPv6 shares with IPv4, and shows how the infrastructure for IPv6 is unique. It discusses in detail some of the Linux protocol facilities that are unique to IPv6.

Note All request for comments (RFCs) referenced in the book are listed in Appendix A. In addition, for the convenience of the reader, copies of all the referenced RFCs are provided in the companion CD-ROM.

1.1 Introduction

The Linux operating system was originally conceived and written by Linux Torvalds with the assistance of many people all over the world. It was intended to be an open source replacement for Unix and is compatible with Unix in almost every respect. The TCP/IP stack is implemented as part of the Linux kernel. It is also compatible with all the applicable Posix standards. Almost all programs written for Unix readily port to Linux without modification. Like the rest of the operating system, the TCP/IP stack was specifically designed for compatibility. In addition, it is completely interoperable with other TCP/IP implementations. It provides a socket interface that is compatible with Berkeley sockets. The code in Linux TCP/IP is very stable, and this is demonstrated by how few changes there are to the TCP/IP code in recent revisions.

In this chapter, we will show how the Linux sources are organized and where to find sources that are referenced in this book. Then, we will introduce a brief history of data communication. We will introduce the OSI seven-layer model and how it forms a basis for examining networking protocols of all types. We will discuss connection-oriented and connectionless protocols. Following this will be a discussion of the difference between local area networks (LANs) and broadband or wide area networks (WANs). In this chapter, we will talk about networking standards and provide a guide for navigating among the various standards bodies.

1.2 The Linux TCP/IP Source Code

The sources in this book are based on the 2.6.0-test10 kernel release with some patches applied. All sources in this book are covered by the GNU Public License (GPL). For a copy of the license, refer to the CD-ROM included with this book or Appendix B, the CD-ROM." Copies of the GNU public license can also be obtained at *www.gnu.org/copyleft/gpl.html*.

The 2.6.0-test10 kernel was the most recent stable 2.6 kernel version available at publication time, and the kernel sources were updated with sources from the Linux IPv6 Development Project known as UniverSAl Ground for IPv6 (USAGI). The USAGI project is where the majority of Linux IPv6 development activity is at the time of publication. However, the official 2.6 Linux kernel releases had not yet incorporated the changes from the USAGI project. There is no official stable release of USAGI available yet for the 2.6 kernel at the time of publication of this book. The most recent stable release of USAGI is 4.1 for the 2.4–20 kernel. Therefore, we decided to go into publication with the 2.6.0-test10 kernel sources patched with the USAGI snapshot dated November 24, 2003. The complete patched kernel source tree is provided on the companion CD-ROM.

In this section, when we refer to *pathnames*, we use the term *linux* to mean the top-level directory in the Linux source tree. For example, if the 2.6 kernel sources are placed in the traditional place for linux kernels, /usr/src, in our case, *linux* would expand to */usr/src/linux-2.6.0-test10*. In this book, we are concerned primarily with the source and header files related to networking and TCP/IP. The TCP/IP sources are in the *linux/net* directory. Most of the core networking source files that are not specific to either IPv4 or IPv6 are in the *linux/core* directory, which contains fundamental networking infrastructure definitions and functions used by both

IPv4 and IPv6 and other protocols. The directory *linux/net/ipv4* contains IPv4 sources and the protocols TCP and UDP. Most of the IPv6 files are in *linux/net/ipv6*.

The network interface drivers are in *linux/drivers/net*. In some cases, there is a separate directory in *linux/drivers/net* for each specific network interface hardware type. Most drivers, like the tunnel pseudo-driver tun.c, are found in the *linux/net/drivers* directory. The generic device initialization functions and definitions in *net_init.c* are in *linux/net/drivers* as well. The companion CD-ROM does not include all the drivers in the Linux source but does have sources for the drivers discussed in this book.

1.3 A Brief History of Data Communication

The purpose of data communication is to transport symbolic or abstract information across great distances. The early beginnings of information transmission go back a long way. From the beginning of written history and before, humans have been working on ways to transmit information across great distances. The early methods used visible smoke or light flashes that could be observed by people watching from a distance. From the start, communication methods required ways to represent information as signals or messages that could be understood by both the transmitter and receiver of the information. All the early formal ways of encoding messages involved breaking down the information into some sort of easy-to-represent pattern. Of course, these early patterns were not transmitted electrically. Instead, the messages were transmitted as timed bursts of smoke or flashes of light that could be observed from a great distance. However, despite the simplicity, this was the beginning of the evolutionary process that led to modern electronic and optical data communication. Along the way, each method of data communication improves something but introduces new problems or challenges. Attempts to find solutions to the problems lead to innovations that are incorporated in the next generation of protocols. Recent data communication methods are the result of an evolutionary process that can be traced back to early and primitive nonelectrical methods.

1.3.1 The Evolution of Data Communication Methods

Modern data transmission methods actually evolved from primitive methods of signaling used since the earliest days of written history. However, the first common example of the use of electrical data communications was the telegraph. The early telegraphs consisted of a simple on and off modulation of a continuous DC current. When pressed, the telegraph operator's key connected this circuit. Pressing the key for alternating short and long lengths of time form what was known as "dots" and "dashes." The dot was a short duration current, and the dash was a longer duration current [ENCBRIT04a]. This system was invented by Samuel Morse in 1832 and came to be known as *Morse Code*. All methods of data communication in use at that time involved manual intervention. Telegraph operators were trained to key messages onto the wire as fast as possible and to interpret received messages just as quickly. Eventually, however, it became obvious that there was a need for even more rapid interpretation of the telegraph methods.

1.3.2 Coded Transmission and the Printing Telegraph

Once telegraph lines began to stretch from town to town, all over the continent and beyond, there was a need for faster and more reliable message transmission. Most of these improvements involved, in one form or another, what was known as a *printing telegraph*. There were various systems of printing telegraphs tried in the mid- and late nineteenth century. The early printing telegraphs actually printed the dots and dashes on paper as a sequence of lines that could be read by the human eye, but only an experienced telegrapher was able to interpret the printouts.

Most of the early methods used in printing telegraphy required the use of multiple circuits, multiple current directions, or some combination of the two. In those early days, the theoretical advantages of different encodings or of using current directions could not be realized. The methods were not practical because of grounding problems in the single-wire circuits. All this was about trying to increase the capacity of the communications channels. Printing telegraphs needed more capacity but were hampered by the scarcity and expense of multiple wires because inter-urban wiring was expensive and required a separate physical wire for each circuit. These limitations inspired the search for a completely automatic mechanism for data communication. Refer to Table 1.1 for a chronology of some of the key events in the early history of data transmission. For excellent source materials and information about the history of telecommunications, visit the North American Data Communications Museum [HOUSE].

Table 1.1: Events in the History of Data Communication

Year	**Protocol**	**Technology**	**Purpose**
1844	Telegraph	Modulated continuous character	Transmission of messages from operator to operator [NEWTON98a].
1871	Baudot	5-bit code; Represents alphanumeric data. Also known as IA2.	Automatic printing telegraph [ENCBRITa].
1910	Telex	Network of automatic printing telegraphs	Automatic message transmission through a network of teleprinters [ENCBRITb].
1962	EBCDIC	8-bit character encoding	Used originally on IBM mainframes such as the 360/370 series [WIKIPEDa].
1964	ASCII	8-bit character encoding	Used for encoding human-readable characters in digital transmission [ANSIa].
1960s	PCM	Modulation	Digitized voice transmission used in North American T-1 and other digital carriers [NEWTON98b].
1968	IBM Bisync	Half-duplex synchronous	Data transmission [NEWTON98c].
1973	Ethernet	LAN	Local data transmission. Originally invented by Bob Metcalf at the Xerox Palo Alto Research Center [GALENETa]. Standardized as [IEEE802.3].
1974	IBM SNA	Layered protocol network based on	Networking and data transmission among IBM computers and remote peripherals.

Table 1.1: Events in the History of Data Communication

Year	Protocol	Technology	Purpose
		synchronous transmission	This layered networking architecture dominated computer networking before more modern WAN and LAN standards [CISCOa].
1976	X.25	Layered protocol based on synchronous transmission	Interconnect customer equipment through PSN [X.25].

1.3.3 Character Coded Transmission

In 1874, the French inventor Baudot devised a method of transmitting the characters directly over the circuit rather than having a skilled human operator key the transmissions. He came up with what was the first system of encoding Roman characters directly in a telegraph transmission. A skilled telegraph operator would no longer be needed to interpret the dots and dashes in the message. His idea, with a few variations, dominated data communications for almost 100 years. The invention consisted of a 5-bit code called Baudot code. The code used five bits to represent 26 uppercase letters, plus the characters SPACE, CR (Carriage Return), LF (Line Feed), BELL, and 14 punctuation characters. Out of the five bits, two are designated as ***shift bits***, which are used by the receiving machine to differentiate among groups of characters. They are used to differentiate between letters, control characters, and numbers depending on the value represented by the two shift bits. In the first implementations, the Baudot code was punched into paper tape at the sending machine prior to transmission. On the receiving machine it could be directly punched into paper tape and converted into printed text later using a paper tape reader. The tape was punched with holes where a hole represented a digital one and an absence of a hole known as a space represented a digital zero [ENCBRITa].

The earliest methods of automatic data synchronization required Baudot or some other similar character-coding scheme. When it was transmitted, the Baudot bit code was preceded by a start bit and followed by a stop bit. These bits were the first system of synchronizing the sending and receiving machine. They were called ***start*** and ***stop bits*** because the receiving machine was a mechanical device that would start interpreting bits when the start bit was received, and end its interpretation when the stop bit was received. Transmission of each character was separate, and the receiving machine had to wait for the start bit to arrive before beginning to decode the character and would end the decoding of the character when the stop bit was received. On later machines called ***teletypes***, the reading of the character from the transmission line was done with an electro-mechanical system. This device consisted of a rotating wheel connected by an electrical clutch to a continuously rotating motor. Once a start bit was received, the clutch would engage, the wheel would start rotating, and it would "read" each bit until the stop bit was received, at which point the clutch would disengage. The machine would sit idly humming away waiting for the beginning of the next character. This method became known as ***asynchronous data transmission***. Some of you who have actually seen an old teletype machine will notice that the data rate is slow enough that the character reception can be observed. Notice that the wheel starts rotating with the beginning of the train of bits, and stops at the end. As each bit is

interpreted, the machine aligns a group of electromagnets to rotate the print wheel to the correct position to print the received character after the stop bit is received and the wheel stops turning.

1.3.4 Measuring Data Communication Speeds

For many years, data communication speed was measured in baud rate rather than bits or words per second. The origin of this word has an interesting history. The word *baud* is actually short for Baudot, who invented the 5-bit code that became known as Baudot code. This code was used for many years in telegraphy. Baud rate is a way of measuring the speed of asynchronous (character-oriented) data transmission. People often incorrectly use this term as if it were synonymous with character transmission rate provided by any type of data transmission. Baud rate is actually the number of 0 to 1 state transitions per second. This concept does not directly calculate out as characters per second. Characters per second can't be determined precisely by dividing the baud rate by the number of bits per character. This is because baud rate includes overhead due to the presence of the stop and start bits. To translate baud rate to character rate, you have to take into account the number of bits used to encode each character. For example, data transmission was at one time limited to at most 110 baud, which was the fastest rate at which the electro-mechanical teletypewriters could receive the transmission. However, the maximum character rate was really much less than 10 characters per second because the 110 baud number includes overhead for the start and stop bits in addition to the 8-bit characters.

1.3.5 Data Transmission Over the Telephony Voice Network

The telegraph was in common use for less than 20 years when the telephone was invented in 1876. Then, for more than 100 years from 1876 until at least 1976, the primary user of long-distance bidirectional information exchange of any type was voice telephony. The familiar system of telephones and voice transmission is known as the Plain Old Telephone System (POTS), where the voice signal was transmitted by analog modulation of a carrier tone. This system is based on assigning a real twisted pair of wires, or later a conceptual circuit, at the start of each telephone call and maintaining the circuit while the call is in progress. In the earliest systems, these circuits were real pairs from the bundle or trunk and the circuits were established by a human operator in the telephone company central office, who would physically connect the phones based on the request of the caller[WIKIPEDb].

Late in the nineteenth century and early in the twentieth century, the number of telephones started to grow rapidly. There were too many phone calls to establish the connections by hand. A faster method was needed to construct the circuit connecting the phones engaged in the call. The improved method was a mechanism to make and break the circuit repeatedly, the dial or pulse generator. When a caller dials the phone, a sequence of pulses is emitted called *dial pulses*. Now, instead of patch panels, the circuit can be created automatically by electromechanical equipment at each switching point between the two phones. This switching point is called a *central office*. The request to construct the circuit begins at the central office nearest to the phone initiating the call. Relays at the central office forward the request to the next switching station by coupling the input circuit to a specific outgoing circuit, extending the circuit to the next switching station, and eventually to the called party. The first telephone switch was known as the Strowger Switch after Almon Strowger [RBHILL].

1.3.6 Multiplexing Data Communication Channels

It is interesting to note that early developments in nineteenth-century mathematics form a basis for twentieth-century data communications theory. This is not a single event that can be placed in the timeline presented in Table 1.1, but instead is a critical evolutionary step toward the development of mid-twentieth-century developments in data communication. The theory shows that a waveform can be thought of as a periodic continuous function. This complex waveform or function can be represented by a series of trigonometric functions called the *Fourier series*. The nineteenth-century French mathematician, Joseph Fourier (1768–1830), developed this idea during his search for a solution for a partial differential equation describing heat diffusion. This is called the Fourier series and allows a continuous function or waveform to be thought of as component sinusoidal functions. The *Fourier transform* converts the complex function into its components. Application of Fourier transform allows the breaking up of available bandwidth of a carrier to individual components or channels [GALENETb].

Eventually, as technology progressed, voice traffic was aggregated onto common lines called *trunks*, named after the cables of bundles of twisted pairs of wires. These new trunks were using Frequency Division Multiplexing (FDM). With the FDM method, an analog broadband carrier is modulated with separate bands of 3-KHz voice channels with a 1-KHz separation. Along with FDM came automatic switching equipment that instead of using electromechanical relays could electronically switch circuits using Dual-Tone Multi-Frequency (DTMF) tones generated by "touch-tone" phones.

As computers and peripheral equipment became more spread out, there was increasing interest in data communications. A need was becoming prominent for a method of data transmission that could use the ordinary and ubiquitous POTS lines. If telephone lines could be used for data transmission, data could be sent from or received anywhere there was an available telephone connection, and telephones were becoming ubiquitous. The first modems were developed in the late 1960s. They modulated the 3-Khz voice band with digital data. Modems permitted data to be transmitted through long distance lines with inductive loading characteristics. The digital signals could then be sent over lines originally intended only for transmission of analog voice signals. The first commercially available modems for use with POTS could transmit at only 300 baud. Modems got their name because they originally were responsible for the ***mod***ulation and ***demod***ulation of the analog frequency with digital data. Modems modulate the voice band with data that is encoded as sequences of 8-bit characters. This encoding could be in various methods, but in modern use is almost always ASCII, which is very similar to the 5-bit Baudot code used since the telegraph era [ENCBRITc] [ITUTTV21].

Chapter 2, "Broadband Networking Protocols of Yesterday and Today," includes more detail about synchronous and asynchronous data communication methods in the discussion about broadband and WAN protocols. We will see how asynchronous data transmission could be sent digitally directly across carriers without using a modem to convert it into an analog signal. However, because for many years direct digital lines were not available everywhere, modems continued to be widely used. In addition, new modem standards became available and allowed for faster transmission of data.

1.4 The OSI Seven-Layer Network Model

The Open System Interconnect (OSI) seven-layer model [OSI7498] was first specified in the early 1980s. Although neither traditional nor modern networking protocols fit neatly into this model, it has been the common framework to discuss and explain networking protocols for over 20 years. The layered model is often called a *stack* because each layer in the sending computer in effect has a logical relationship with the corresponding layer in the receiving machine. See Table 1.2 for an illustration of the seven-layer model.

Table 1.2: The OSI Seven-Layer Model

Layer Number	**Layer Name**
7	Application
6	Presentation
5	Session
4	Transport
3	Network
2	Data Link
1	Physical

1.4.1 The OSI Lower Layers

The lowest layer, Layer 1, is the *physical* layer. It is concerned with the lowest level of detail about data transmission. This layer worries about issues such as how to indicate the presence of a one bit versus a zero bit on the physical media. It also deals with electrical signal levels and other details related to physical transmission. The next layer up, Layer 2, is the *data link* layer. The data link layer is primarily responsible for establishing and maintaining connections or providing connectionless service. It is concerned with how the two endpoints will establish a communication. It also deals with framing or how to differentiate user or payload data from control data. It contains a way for machines to identify each other, generally called *station IDs* or *Media Access Control* (MAC) *addresses*. The next layer up, Layer 3, is the *network* layer. It is responsible for how nodes on the network are named or addressed. It is concerned with network topology and how to route packets from one node to another. As with the data link layer, the network layer has address information, too. However, network addresses are more abstract; they have to do with network topology and at least theoretically are not tied to a particular physical interface. The next layer up, Layer 4, or the *transport* layer, provides a way for data to be collected or aggregated for passage across the network. In addition, the transport layer provides a simple programming interface to users at higher layers so they don't have to deal with the network details of Layer 3 or lower. In the world of TCP/IP, we think of the transport layer interface as providing two types of interfaces. The first is a streaming interface where the user can send and receive data as a continuous stream of bytes. The other interface provides a chunked or datagram interface to the user where the user must break up the data into discrete packets before sending.

1.4.2 The OSI Upper Layers

The higher layers are often thought to be part of the application. Layer 5, the *session* layer, is primarily about managing access and session control of a user on one machine who wants to access another machine. Layer 6, the *presentation* layer, is for abstract data representation. Data in network representation is mapped to the user's view so the application need not worry how the data looks when it is stored on a different machine. Architectural details of data representation in a particular machine are removed at this layer. The uppermost layer, Layer 7, is the *application* layer. Only pure OSI-compliant networking protocols think of this as a separate layer. It is important to point out that the upper layers—Layers 5, 6, and 7—are not part of the TCP/IP stack. (See Chapter 3 for a discussion of the upper layers and their relation to the TCP/IP stack.)

1.5 Connection-Oriented and Connectionless Protocols

WAN or broadband protocols are typically connection oriented. WANs evolved from the exploitation of spare capacity of the Public Switched Telephone Network (PSTN). The public network was originally intended to carry voice traffic; therefore, the connections are actually virtual circuits that were intended to provide a path for voice traffic between telephones. Before individual packet-switching service was available through the PSTN, each time data was transmitted between two machines a provisioned connection had to be set up first. The connections used in broadband networks are called *virtual circuits* (VCs). There are two types of VCs: permanent virtual circuits (PVCs) are provisioned and maintained as static connections. Switched virtual circuits (SVCs) are set up through signaling between the end systems when a call or connection is requested. In the TCP/IP stack, these concepts are incorporated in the transport layer rather than in the data link layer.

1.6 Broadband Networking Versus Local Area Networking

For the purposes of this book, the term *broadband* is used synonymously with the term *wide area network* (WAN). In both the popular press and the engineering press, the term *broadband* is generally used to differentiate the slower POTS-based dial-up line from a faster digital subscriber line (DSL) or cable modem. There is one important differentiating factor between local area networks (LANs) and WANs. Every machine on a LAN can see every other machine. However, on WANs, the connections are virtual circuits. They are set up either specifically as point-to-point connections between two machines or as multicast connections of one-to-many. LANs and WANs can also be differentiated by the role played by the data link layer of the OSI model. Chapter 2 contains a discussion of the OSI model, and Chapter 3 shows how the OSI model fits in with the TCP/IP protocol. The data link (DL) layer for a LAN worries about framing, transmission, and receiving. The data link layer for WANs is far more complex. It is also concerned with connection negotiation and management as well as providing configurable degrees of reliability or speed.

1.6.1 Local Area Networks

LANs are newer than WANs. The concept of LANs was developed about 20 years ago. On a LAN, each machine can see every other machine. Some type of low-level machine identification scheme is used to identify each computer. In addition, LANS include a method of arbitrating

among machines that attempt simultaneous access to the network. The first and still the most popular LAN is Ethernet. From the standpoint of TCP/IP and some other data communication stacks, the link layer is far simpler in Ethernet and other LANs than it is for WANs. This is because there is no requirement for connection-oriented service. However, if the particular network configuration includes packet filtering, bridging or switching, or address translation, these capabilities will add quite a bit of complexity to the data link layer. Almost always, the data link layer for Ethernet consists only of a framing layer and the MAC header. All of Ethernet's complexity is hidden in the physical layer (PHY). It is at the physical layer where error detection and collision avoidance occur.

Generally, most LAN reception and transmission is handled at the device driver level. Incoming packets are written directly in a circular buffer by the DMA capability of the Ethernet interface. Outgoing packets are queued at the Ethernet chip for transmission. There are more complex LANs such as WIFI 802.11. Although these protocols are more complex than Ethernet, the details are hidden in the physical layer, and the data link layer provides a fairly simple Ethernet-compatible interface. Of course, authentication and authorization complicates the picture, but once a user is validated, the interface presents a simple LAN type interface where each machine on the immediate LAN can see all the other machines.

1.6.2 Wide Area Networks

In this book, we look at things from the viewpoint of TCP/IP. Essentially, we consider protocols other than TCP/IP from two points of view. We will see how some historical protocols contributed essential technology that later became in corporated in TCP/IP. Other protocols are still commonly used for WAN data transmission and often are interfaced to TCP/IP.

WANs are concerned with using capacity on the public carrier networks. There are many methods to interface computers to WANs, from simple modems, DSL, and cable modems, up to high-speed optical interfaces such as OC192. Essentially, however, there are two types of WAN interfaces. The first is a direct point-to-point connection through a dedicated or private link. An example would be a dedicated physical twisted pair or a leased line. These will be interfaced as a network interface driver with hardware support for the interface. The other more complex type of WAN interface involves a complete separate protocol suite with elements of the protocol provided to manage negotiated bandwidth and data traffic classes. These protocols are for use in a public network where data traffic is merged with other rate-payer's data or where data packets are submodulated on a shared carrier along with other data or voice traffic. An example of this protocol would be Frame Relay or ATM. Interfacing protocols such as these to TCP/IP involves one or more network interfaces without low-level hardware support. Examples of these interfaces might be IP over Frame Relay, or Classical IP over ATM (CLIP). Some examples of WAN data communication protocols are discussed in more detail in Chapter 2.

1.7 Packets and Frames

When discussing computer networking, there is confusion between when to call a chunk of data a *packet* and when to call it a *frame*. Often, when IP-based networking is discussed over Ethernet,

the difference between these two terms is blurred even though there is a difference between them. When discussing data transmission at the data link layer or the physical layer, the term *frame* is used to describe a unit of data. A frame is thought to consist of a beginning sequence of characters or bits defining the start of the frame. This is followed by a sequence of characters or bits containing the payload data and followed by a sequence of characters or bits defining the end of the frame. When discussing data transmission at the network layer or above, we consider the unit of data transmission a packet. A packet is a data chunk following a header, and we think of the header as encapsulating the data. The header typically includes a length field because packets are generally considered variable length. In contrast to a packet, a frame is a data unit delineated by a starting sequence of bits or flags and terminated by a stream of specific bits or flags. Another important difference is that a packet is intended to be processed by higher-level software, and a frame is intended to be processed by lower-level hardware.

1.8 The Digital Data Rate Hierarchy in The Public Network

Most of the protocols included in this chapter are for transmitting data across the public network. As discussed earlier, the public network evolved from early hardwired voice circuits and evolved over many years to be able to provide a variety of services and different rates. Originally, a separate twisted pair of wires was used for each voice channel. In the mid-twentieth century, the demand for voice circuits rose rapidly and telephone companies looked for a way to combine voice channels in trunk lines without having to use huge cables consisting of bundles of twisted pairs. Voice channels were multiplexed using analog methods across a single modulated broadband carrier using a method called *frequency domain multiplexing* (FDM). With FDM, each voice channel is modulated on a specific frequency band within the broadband carrier using frequency modulation (FM) techniques. This technique continued to be used for about 20 years until it became apparent that it was not the most efficient use of broadband channel capacity. Moreover, FDM required expensive analog switching equipment in the telephone company's switching stations and therefore was not scalable to ever-increasing demands for increased circuit carrying capacity [EDWAR00a].

Later, in the 1960s and 1970s, the public network was converted from analog to digital. Voice phone calls were transmitted over individual channels modulated using *pulse code modulation* (PCM) over a base band carrier. The voice information is modulated and divided into frames where the local analog loop terminates at the phone company's central office (CO). Each voice channel can be transmitted using time division modulation (TDM) techniques. It was known that to maintain minimum fidelity for voice transmission, each channel requires 3 kHz of bandwidth. It was determined that 64 K bits per second was required to transmit a single voice line. Therefore, the fundamental unit of bandwidth for the public network is 64 KB, and all digital data rates are calculated in units of 64 KB of capacity. Table 1.3 shows the hierarchy of digital channel capacity carried across electrical networks in North America. Table 1.4 shows the equivalent hierarchy in use in Europe. Table 1.5 shows the Sonet and SDH Optical Hierarchy [MCDYSO99].

Table 1.3: TDM Digital Hierarchy in North America

Name	Multiplexing Level	Number of Voice Channels	Rate in Bits per Second

Table 1.3: TDM Digital Hierarchy in North America

Name	Multiplexing Level	Number of Voice Channels	Rate in Bits per Second
DS0	0	1	64 K
DS1	1	24	1.544 M
DS2	2	96	6.312 M
DS3	3	672	44.376 M

Table 1.4: TDM Digital Hierarchy in Europe

Name	Multiplexing Level	Number of Voice Channels	Rate in Bits per Second
E0	0	1	64 K
E1	1	30	2.048 M
E2	2	120	8.448 M
E3	3	480	34.368 M

Table 1.5: Sonet and SDH Optical Hierarchy

SONET	SDH	Line Rate in Mbits per Second
OC-1		51.84
OC-3	STM-1	155.52
OC-12	STM-4	622.08
OC-48	STM-16	2488.32
OC-192	STM-64	9953.28

Carriers also provided specially provisioned twisted pairs to customers for carrying digital data. Each of these lines, called *leased lines*, was specially provisioned to carry more than the 64KB of bandwidth that was typical for a single channel intended for voice. Originally, there was no standard to determine the bandwidth and capacity for these leased lines. Eventually, these nonregulated channels were replaced with standard channels based on available standard rates.

1.10 Summary

This chapter gathered some basic concepts and history that will help to build a basis of understanding of data communication, and TCP/IP in particular. We presented an introduction to data communication. Some of the early history of data communication was discussed. We learned how modern data communication was inspired by the need for a printing telegraph. We saw how it evolved with a series of steps up to modern high-speed optical switched networking. In Chapter 2, we will see more about non-TCP/IP standards that form the evolutionary basis for modern data communications. We will learn how the OSI model serves as a basic framework to discuss computer networking. In Chapter 3, we will apply the knowledge to TCP/IP. Chapters 4, 5, and 6 cover the TCP/IP infrastructure in Linux in detail.

Chapter 2: Broadband Networking Protocols of Yesterday and Today

In Chapter 1, "Introduction," we discussed the basics of data communication and presented the OSI seven-layer model. We illustrated the difference between broadband and LAN protocols and covered networking standards. Now, to fully appreciate the TCP/IP protocol suite and how it came to be so widely used, it is helpful to discuss other networking protocols that influenced the development of TCP/IP. Some protocols in this chapter are included because they are often used to interface to TCP/IP when we want to send IP datagrams across a broadband or public network. Others are included because they are significant in a historical context and have concepts that were later borrowed by TCP/IP. Still others are included because they extend the reach of IP packets into broadband realm and allow IP packets to be forwarded over the public networks.

2.1 Introduction

In this chapter, we extend our discussion on basic data communication begun in Chapter 1. We will discuss the difference between asynchronous and synchronous data communication protocols. From a historical context, we will introduce different types of synchronous protocols and show an example of how they are used in the X.25 stack. High Level Data Link Protocol (HDLC) will be covered, too, because it is the basis of both early and modern protocols using electrical transmission methods. We will see how concepts used as part of TCP connection management evolved from X.25 and HDLC. Frame Relay is important, too, and we will look at it to see how it differs from X.25. The last but far from the least important protocol is Asynchronous Transmission Mode (ATM). It is the most recent of the broadband protocols and is important because it forms the basic infrastructure in the public telephony network that is the basis for most voice and data communication. In addition, ATM provides a common point of interface for sending TCP/IP traffic to the world outside our LANs.

The material in this chapter will include some important concepts that should help in the detailed examination of Linux TCP/IP in later chapters. It is important to note that most of the broadband protocols we cover are primarily concerned with providing reliable or connection-oriented service. As we will see, when we go into more detail in the chapters on TCP/IP, IP traffic was intended to be carried over connectionless networks. TCP, which runs over IP, is depended on to provide a connection-oriented transport for most of the common application layer protocols. However, we will see by examining the lower layer protocols in this chapter that the connection-oriented protocols here are analogous to TCP's implementation of connection management.

2.2 Connection-Oriented and Connectionless Protocols

WAN or broadband protocols are typically connection oriented because they provide sequenced reliable delivery service. As we saw earlier, WANs were intended to use excess capacity in the Public Switched Telephone Network (PSTN). The PSTN evolved from the earlier discretely wired telephone system that was in use well into the last century. The virtual circuits have to maintain the audio signal quality, and they work the same way as a physically connected pair of telephones does. The same public network, originally designed to carry voice traffic, evolved to

a mixed traffic network where it is asked to carry data simultaneously as it carries voice data in connections implemented as virtual circuits. This orientation to voice traffic naturally led to the development of connection-oriented protocols, which are also circuit oriented. The simpler of the connection-oriented or WAN protocols use the network in the way for which it was designed: to carry voice channels over the public network.

In a similar fashion to circuit-based voice traffic, connection-oriented pro tocols work over virtual circuits (VCs). The part of the protocols that establish the circuits is implemented at Layer 2 or higher in the protocol stack. (Refer to Chapter 1 for a description of the OSI seven layer model.) The VCs are either configured at startup time or are dynamically set up. When they are established at startup time, they are called provisioned virtual circuits (PVCs). When they are set up dynamically by a signaling protocol, they are called switched virtual circuits (SVCs). The signaling methods are actually implemented as separate protocols that negotiate the connection between pairs of switches. Once they are set up, the VCs are actually full-duplex paths between pairs of nodes.

LANs can be contrasted to WANs by the way in which the neighboring machines are connected. Machines on WANs don't really have neighbors; every connection between two nodes is either pre-provisioned or separately negotiated. However, machines on LANs physically near each other don't need to run special software to negotiate a connection with other nodes before they can talk to each other. On a LAN, every machine can see all its neighbors. The machines on a LAN are all peers; every machine has the equal right to transmit to another machine. The physical layer arbitrates in some fashion between the transmitting machines based on their link layer addresses, and the upper layers are responsible for filtering out unwanted packets.

2.3 Broadband Networking Versus Local Area Networking

In this chapter and elsewhere in the book, we use the terms *broadband* and *WAN* as synonyms. Although not precisely correct, this is the terminology in common use at the time of publication. WAN network interfaces faster than analog modems are called "broadband" connections. As discussed in earlier sections, the simpler broadband or WAN protocols with some exceptions establish point-to-point connections. They rely on higher layer protocols to provide multicast or broadcast capability. In contrast, LANs are democratic. All the machines on a LAN share the same chunk of bandwidth. They can all talk at once or at least try to talk at once. They can all see each other's transmissions. Therefore, LANs provide a mechanism at the physical layer to negotiate access to the physical network to make sure that there is only one speaker at a time. In general, LANs have a fairly complicated physical layer but very simple data link layers. LANs will checksum packets to make sure that bit-level communications are okay. Although packet integrity is usually guaranteed, errors will result in dropped packets.

WANs usually have relatively simple direct serial interfaces at the physical layer. Higher layers, usually the data link layer, have to do all the work of arbitrating bandwidth and negotiating the connections. Therefore, when compared to LANs, WANs have simpler physical layers but more complicated data link layers. The data link layers will compensate for transmission problems at the physical layer provided sufficient bandwidth is available.

From the viewpoint of TCP/IP, if the data link and physical layers do their jobs, the details of data transmission are mostly transparent to Layer 3, the network layer, and the layers above Layer 3. If the lower layers don't do their job, the result will be dropped packets. If the data transmission is using the TCP transport protocol, effort will be made to retransmit dropped packets at the lower layers. However, as we will see in later chapters, TCP can't entirely compensate for low-quality links.

2.3.1 Local Area Networks

LANs are newer than WANs. The first and still the most popular LAN is Ethernet. The link layer is far simpler in Ethernet than it is in broadband protocols. A basic Ethernet Layer 2 interface consists only of a framing layer and the Media Access Control (MAC) header. In its simplest form, the MAC header contains the source address, the destination address, and the protocol number of the payload. The complexity of the LANs is hidden in the physical layer (PHY). LANs have the characteristic where each machine can see the transmissions of all the other machines that are directly reachable. There is no allocated bandwidth or channels. Instead, LAN protocols provide a way to negotiate access to the network by allowing only one machine to transmit at a time. There have been various schemes and we don't intend to cover them all here. However, the most popular LAN, Ethernet provides a method called Carrier Sense Multiple Access with Collision Detect (CSMA/CD) where each machine waits for a clear carrier before starting to transmit a packet. In addition, each machine generates a Frame Check Sequence (FCS) code, and the recipient machine will drop the packet if the FCS is bad.

Generally, all the transmission details, including collision detection and error detection, are done in hardware or low-level firmware. This hardware is the physical layer and it is hidden from the data link layer and all other layers. With most Ethernet interfaces, the data link layer is responsible for receiving the incoming packets, which are placed directly in a circular buffer by the Direct Memory Access (DMA) capability of the Ethernet interface. Incidentally, many Ethernet interfaces can place the packets arriving sequentially in noncontiguous locations and this is known as *scatter-gather* capability. When available on an Ethernet interface, Linux makes use of scatter-gather by minimizing expensive copying of packets received from Ethernet interfaces. Similarly, outgoing packets are queued at the Ethernet chip for transmission, and if the interface has scatter-gather capability, these packets are transmitted directly from a linked list so they don't have to be copied separately by software into a separate buffer.

Most LAN protocols, although they might be far more complex than Ethernet, present a fairly simple Ethernet-compatible interface at the data link layer. This type of interface is specified by the ISO 8802 series. Wireless protocols also provide an Ethernet-like interface to the data link layer. They do have Authentication and Authorization (AA) capability, but this function is invisible to the upper layers. Essentially, once a user is authenticated, from the TCP/IP point of view, the wireless LAN provides a simple Ethernet type interface. Each machine on the immediate LAN can see all the other machines that are directly reachable.

2.3.2 Wide Area Networks

Broadband interfaces generally involve public carrier networks, including everything from DSL and cable modems to high-speed optical interfaces such as OC192. Essentially, from the

standpoint of this book, there are two types of WAN interfaces. The first is a long-range, reliable, point-to-point link through a dedicated or private line. An example would be a dedicated-physical twisted pair or a leased T1 line, and although of less interest, this type of interface will be discussed briefly in order to build an understanding of basic data transmission methods. The other more complex type of WAN interface attaches to a public network where data traffic is merged with other rate payers' data or where IP packets are submodulated on a shared carrier along with other data or voice traffic. This second type of interface includes IP over Frame Relay, (IPOA), or DSL, and any other interface where IP traffic is carried on another network type.

2.4 Asynchronous Versus Synchronous Data Transmission

It is important to differentiate between asynchronous and synchronous methods of data transmission. Asynchronous data transmission has more overhead and doesn't scale to higher rates the way synchronous methods do. The early methods discussed in Chapter 1 on the history of data communications were done by sending sequences of characters in a sequential linear fashion. If transmission is done this way, every character transmitted adds to the overhead. Each character has framing bits adding to the amount of data that needs to be transmitted. In addition, with asynchronous methods, there are no synchronizing external clock sources, nor are there are any synchronizing pulses embedded within the data stream. Each character is separately framed with start and stop bits. The receiving station must resynchronize on each character, which is why the maximum speeds are limited.

In contrast, synchronous data transmission methods don't frame every character with individual start and stop bits. Instead, the characters are grouped together in sequence into frames, and the frames are transmitted as a stream of bits or characters. Each frame consists of a sequence of beginning bits or characters, followed by a data payload and then followed by a terminating sequence of bits or characters. The overhead is far higher for asynchronous methods and could be as high as 25% depending on the character length and how many framing bits there are for each character. However, the overhead for synchronous protocols is far lower and could be as little as 5%.

2.4.1 Flow Control and Reliable Transmission

Flow control and reliable transmission are really two topics. In this section, we discuss these topics in the context of low-level transmission control. At the lower layers, we are concerned with how to provide acknowledged frames for error recovery and sequenced delivery. In addition, we maintain queues at the transmission side to accumulate data waiting to be sent and provide an even flow of information. Before we begin, it is important to note that TCP/IP does not require either reliable or sequenced packet delivery service. TCP/IP is designed to work with LANs that don't implement retransmission in the case of dropped packets at the lower layers.

The method of low-level flow control used is different with simple character-oriented protocols than when it is used with faster synchronous-oriented protocols. Character-oriented protocols in their simplest form provide the most basic method of flow control. The sending station must wait for an Acknowledge (ACK) from the receiving station before sending the next character. This mechanism is known as *stop and wait acknowledgment*. With this method, it is impossible to fully use the channel's bandwidth because one station sometimes has to send empty sync

characters or otherwise mark time while waiting to receive an acknowledgment of an earlier frame. This problem is solved with bit-synchronous protocols by using a technique called *sliding windows*, which is far more efficient. The sliding windows technique has less overhead by using multiple acknowledgments where one ACK can collectively acknowledge multiple incoming frames. After sending the first frame, the sending station can continue to send additional frames without stopping to wait for the receiving station to acknowledge the first frame. An example of sliding windows is explained in more detail in Section 2.8 in this chapter. The TCP protocol also implements a version of sliding windows, and Chapters 8 and 11 include how it is implemented in the Linux TCP/IP stack.

The data link layer typically provides a link status indication to be used with flow control. Queues of frames waiting for transmission are maintained below the network layer but above the data link layer. When the higher layer has a packet ready, it is added to the end of the queue. When the lower layer is ready to transmit a packet, it removes the next packet from the bottom of the queue. This ensures that higher layers continue processing independently from the lower-layer link status.

2.5 Synchronous Data Transmission

In the previous section about asynchronous data transmission, we discussed how there is higher overhead associated with asynchronous character-oriented methods. This is because each character must be acknowledged separately to maintain reliable transmission. In addition, there is higher overhead associated with asynchronous transmission because each transmitted character is preceded by start bits and terminated by stop bits. However, with synchronous forms of data transmission, the overhead is far lower. Instead of sending each byte of data as an individual transmission, the data is gathered into frames and transmitted together as either a stream of bits or a stream of characters. Each frame is preceded by a flag sequence and is followed by a flag sequence. Because of less stopping and less extra bits, synchronous methods have far less overhead than asynchronous methods do.

2.5.1 Synchronous Character-based Data Transmission

The primary advantage of synchronous methods of data transmission is that they have lower overhead when compared with asynchronous methods, and therefore are better suited for high-speed transmission. However, to understand the more recent high-speed methods, it is important to take a moment and look at an earlier method. The first binary synchronous protocol in common use was developed by IBM and was known as *BISYNC*, or BSC. This protocol was the basis of what was to become the Systems Network Architecture (SNA) protocol suite. Although it was only available as part of IBM's product line at the time, SNA was an early attempt for diverse computers and telecommunications equipment to exchange data using one basic standard.

As with asynchronous character-oriented protocols, BISYNC and other character-oriented synchronous protocols must have a specific character coding specified for the particular protocol. The data is framed by using special characters called *control characters*. The control characters are used to indicate the start of a frame, the header and control information, and the end of a data frame. Table 2.1 shows the specific control characters used with the BISYNC protocol. Table 2.2 shows the framing used with BISYNC and how the control characters are used for framing. The

start of a frame is indicated by two consecutive *syn* characters; the presence of the *stx* character in the data stream indicates the beginning of payload data. The *etx* character is used to terminate the frame. The character coding method used with BISYNC was called EBCDIC, which was used with IBM and IBM-compatible products. Other character-oriented protocols used American Standard Characters for Information Exchange (ASCII). The ASCII character set is still the basis of almost all 8-bit character representations used for representing European languages.

Table 2.1: BISYNC Control Characters

Character	**Name**	**Purpose**	**ASCII Value in Hex**
ack	Acknowledge	Acknowledges a frame.	06
etb	End of data block	Terminates a block of data.	17
etx	End of text	Terminates a block of data.	03
eot	End of transmission	Indicates end of transmission.	04
enq	Enquiry used for polling by controlling station	Used by polling station to query for a response.	05
nak	Negative acknowledge	Acknowledges a frame to indicate errors were received.	15
dle	Escape	Tells the receiving station that the next character is a control character.	10
syn	Synchronous idle	Maintains synchronization without sending data.	16
soh	Start of header	Indicates start of the header.	01
stx	Start of text	Indicates the end of the header and the start of the data.	02

Table 2.2: BISYNC Frame

SYN	SYN	SOH	Header	STX	data....	ETX	BlockCheck

One of the problems with character-oriented protocols is the lack of data transparency. The ASCII and EBCDIC character sets represent a subset of all the bit combinations in one byte. There is a problem when we must communicate data that includes nonprintable characters. We need a method to encode data that happens to contain bit patterns that correspond to one of the control characters. Originally, the character coding schemes were only intended to represent actual characters corresponding to printed text. Unfortunately, not all transmitted data consists of data that can be represented in characters from a particular coding scheme. When there is a need to exchange unrestricted binary data, which is most of the time, the data must be encoded as hex characters or some other method that only requires characters from within the encoding set used for the protocol. When data bytes are represented randomly in a set like ASCII, occasionally an individual byte of data can be identical to a control character. In this case, character-oriented protocols must use a scheme called *escaping*. Each of the common character encoding schemes contains an escape character. The escape used in EBCDIC was the *DLE* character. In ASCII, it is the same as the familiar escape character still found on everyone’s computer keyboard today. The purpose of the escaping is to indicate to the receiving station that the byte immediately

following the escape should be interpreted as data and not as a control character. This prevents the receiving station from dropping the connection or making some other misjudgment when it encounters a byte in the data stream that it mistakes for a control character in the data stream. If the receiving station gets two escape characters in sequence, it interprets these as a single escape.

2.5.2 Count-oriented Synchronous Protocols

Count-oriented protocols solve the data transparency problem inherent in character-oriented protocols. As discussed previously, the problem occurs when the data field of a frame contains a byte of data that happens to be identical to one of the control characters normally marking the end of frame or beginning of a control message. The count-oriented protocols solve the problem by inserting a length field before the payload data. In this way, all possible binary types of data could be included in the user payload data. Overhead is reduced because, unlike character-oriented methods, the transmitting station does not have to sift through all the data in the transmission inserting *escape*s before each occurrence of a binary data character. Count-oriented protocols introduced a new type of framing with a framing header that includes a length field. Incidentally, the type of framing used in these protocols is the most popular; it forms the basis for most modern data framing techniques used in electrical transmission. As we will see in Chapter 3, "TCP/IP in Embedded Systems," TCP/IP makes extensive use of this framing technique. A side effect of the count-oriented protocols is that since they have a header containing a length field, they can have variable-length frames, where the other methods in this chapter have fixed-length frames.

2.5.3 Bit-oriented Synchronous Transmission

In this section, we discuss a family of protocols that are closer to the WAN protocols used in modern electrical and, to some lesser degree, in modern optical networks today. The growth in demand for bandwidth in the late 1970s and 1980s necessitated the need for better data communication protocols. Both count-oriented protocols and character-oriented protocols have disadvantages when they are used with higher-speed data transmission. Both types of protocols required the use of a character encoding scheme such as EBCDIC or ASCII as well as the escaping technique discussed earlier. Particularly when implemented in the hardware available when the protocols were developed, the count-oriented and character-oriented protocols were inherently slow. Bit-oriented transmission methods addressed the performance issue by using a technique called *bit stuffing*. Bit stuffing provides a more efficient method for the receiving station to differentiate among flags marking the beginning of a frame, flags for the end of a frame, errors, and user payload data. One of the earlier protocols that used bit-oriented data transmission is called Synchronous Data Link Control (SDLC). SDLC was eventually used by IBM to replace BISYNC as the basis for SNA. SDLC is also called a link control protocol and could be considered a member of the family of the High Level Data Link protocols, (HDLC). Of course, these protocols aren't really "high level." Today, we consider anything below the network layer low-level protocols, but at the time HDLC was first envisioned, most data communication techniques were very low level—they didnuse any framing techniques at all. SDLC and other link control protocols became widely used along with bit-oriented synchronous transmission methods.

2.5.4 Bit Stuffing

One of the challenges faced by data transmission methods is the encoding of binary data. A data transmission consists of seemingly random data that must be processed by the receiving station while it simultaneously detects error conditions along with frame start and end sequences from among the stream of bits. As discussed earlier, other encoding schemes required binary data to be preceded by *escapes* when binary data was to be transmitted as part of the user payload data. Bit synchronous protocols, however, use a method called *bit stuffing* to address this problem. Bit-oriented protocols indicate the start and end of a frame with a sequence of six 1 bits or a hex 7E. In the payload part of the frame, the transmitting machine inserts a single 0 bit after every sequence of five 1 bits. While it is processing the received data, if the receiving station receives five 1 bits and then sees a 0 bit, it simply removes the 0 bit from the data frame. If it encounters five 1 bits followed by a sixth 1 bit, it interprets these as the start or end of a frame because it is equivalent to a hex 7E. If the receiving station sees between 7 and 15 1 bits in order, it flags this as an error condition. When more than 15 consecutive 1 bits are received in sequence, the receiving station interprets this as an idle channel.

2.6 X.25

At one time, one of the data communication methods in common use was SNA, used widely with IBM mainframe computers. However, around the same time, a need emerged for a communication protocol that would make use of the spare bandwidth in the PSTN. This protocol was called X.25. X.25 is an International Telecommunications Union (ITU) specification. It was originally limited to 2 MB per second. In recent times, X.25 is not used very much in the PSTN. However, because of its robust nature, it is still in wide use today over slow but unreliable terrestrial radio connections. In addition, it sometimes is used at higher speeds than originally specified for certain specialized applications. X.25 is important to include in our discussion of foundational data communication protocols. It is the first example of a truly layered protocol in common use in the public network outside of a particular vendor's control. X.25 actually incorporates the first three layers of the OSI model, the PHY, link layer, and network layer. It was designed for low-quality physical transmission lines so it has error control and correction at both the link and network layers. This redundant error checking is really no longer needed for reliable modern optical or even electrical transmission lines. Currently, X.25 is only used to provide reliable data delivery for low-quality wireless links for certain specialized aviation and military communications applications.

X.25 was the first protocol to use the concept of a virtual circuit (VC) for data transmission. The purpose of VCs is to connect customer equipment in one location across the common carrier's PSTN to customer equipment at another location by using negotiated connections that are analogous to voice circuits. VCs can be multiplexed by combining many separate streams of data packets into a single stream over the public network. The X.25 protocol defines a service access point where the common carrier's equipment and the customer equipment meet. X.25 hides the internal function of the PSTN from the customer's equipment. The carrier's equipment is called *data communications equipment* (DCE), and the customer's equipment is called *data terminal equipment* (DTE).

The top layer or network layer of X.25 is called the *packet layer*. The job of this layer is to manage either connections of switched virtual circuits (SVCs) or permanent virtual circuits (PVCs). The packet layer is responsible for establishing and breaking down these connections, and this connection negotiation is also termed *call management* or *signaling*. It is interesting to point out that the terms *signaling* and *call management* come from voice telephony. See Chapter 1 for background on the evolutionary nature of data communication and how it evolved from telegraphy and voice telephony. The packet layer supports two types of packets, control packets and data packets. The control packets are used for call management and the data packets contain user payload data. X.25 also provides an individual packet delivery service. This service consists of transmitting individual user datagrams without first doing connection negotiation to establish a connection or circuit. As implied earlier, X.25 provides for support of PVCs, which are VCs that are provisioned rather than set up dynamically with a signaling protocol. Refer to Figure 2.1 for an illustration of the layers of X.25.

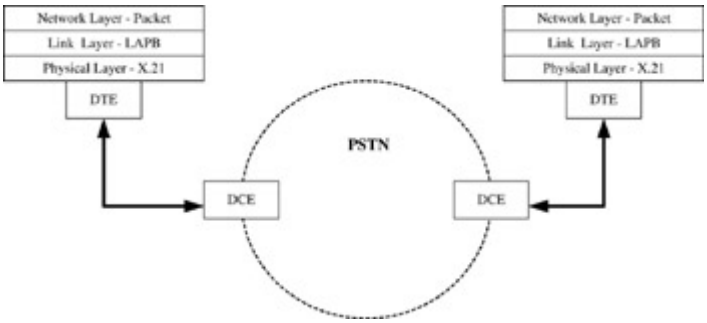

Figure 2.1: X.25.

The middle layer in the X.25 protocol stack is the equivalent of Layer 2, the link layer. It incorporates a protocol called Link Access Protocol Balanced (LAPB). LAPB is another variant of HDLC. The types of HDLC are described in more detail in Section 2.7. In the X.25 protocol stack, the link layer provides both a connection-oriented service and a user datagram service. Even though X.25 contains its own network layer—when X.25 or for that matter, other WAN protocol stacks are interfaced to TCP/IP—the entire stack sits under TCP/IP's network layer and appears as a Layer 2 interface.

2.7 High Level Data Link Protocol

X.25 was the first protocol suite to incorporate a generic Layer 2 protocol for its link layer definition. The generic family of Layer 2 protocols used for WANs are called High Level Data Link Protocols (HDLCs). Originally, when layered protocols were new, "high level" meant that the protocol was dealing with anything more abstract than the electrical details of physical transmission. HDLC is still the basis of many data communication protocols in common use today with the exception of ATM and optical protocols. There are a few variants on the basic theme of HDLC, and most of these variants are governed by ISO standards. However, there are also a few HDLC-type protocols specified by ITU and IEEE standards ISO 3300 and ISO 4335. See Table 2.3 for a list of various HDLC derivative protocols. It is important to point out that although these HDLC variants are similar in the way they work, they do not expect interoperation between variants because the framing details are different.

Table 2.3: HDLC Protocol Variants

HDLC Flavor	Name	Purpose or Use
LAPB	Link Access Protocol—Balanced	X.25
LAPM	Link Access Protocol—Modem	Used with V.42 compression protocols in modems
AHDLC	Asynchronous HDLC	PPP over asynchronous links
BHDLC	Bit synchronous HDLC	PPP over synchronous links
802.2 LLC	Logical Link Control	Basis of data link provider interface in STREAMS
LAPD	Link Access Protocol—D channel	ISDN
SDLC	Synchronous Data Link Control	IBM SNA
LAPF	Q.922 Link Access Protocol for Frame Relay	Frame relay bearer service

2.7.1 HDLC Types and Configurations

This section shows the HDLC protocol in more detail. All the variants of HDLC are functionally similar. They might differ in the precise lengths of the address, control, or FCS fields, but this section is largely applicable to each variant. HDLC supports three types of stations and two types of configuration. The three types of stations are:

Primary: Controlling station on the link.

Secondary: Slaves to primary stations.

Combined: Act as either primary or secondary stations.

In addition to the three types of stations, HDLC specifies two types of configurations. The configurations can be thought of as modes of service. The two types of configurations are:

Balanced: In a balanced configuration, there are two stations connected with a point-to-point link. In this configuration, each station acts as a combined station.

Unbalanced: With the unbalanced configuration, there are two or more stations. One station acts as a primary station. Two or more other stations act as secondary stations.

Figure 2.2 illustrates the unbalanced configuration, and Figure 2.3 shows the balanced configuration. With the balanced configuration, the stations are peers. Each station can send both commands and responses. The balanced configuration is used primarily for point-to-point links. In contrast, with the unbalanced configuration, the primary station is the controlling station on the link and the secondary stations are slaves. The primary station is the only one that can send commands, and the secondary stations answer with responses.

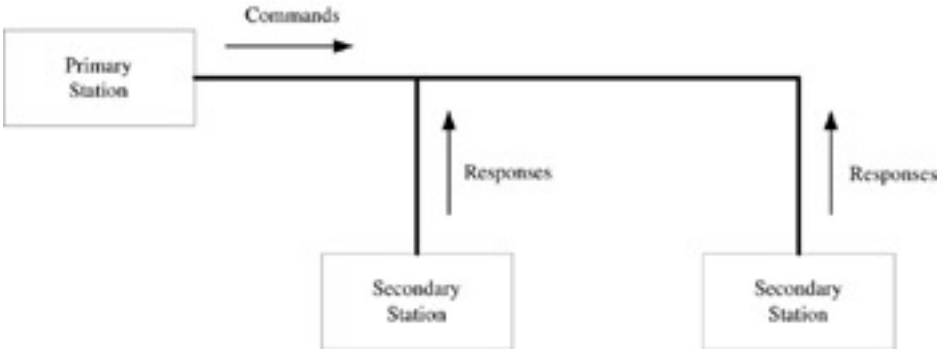

Figure 2.2: HDLC unbalanced configuration.

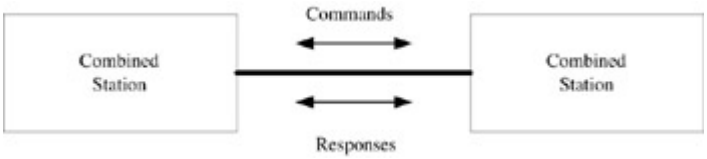

Figure 2.3: HDLC balanced configuration.

2.7.2 HDLC Framing

As we have seen, there are many variants of HDLC. The HDLC framing detailed in this section shows framing that is generally common to all of the HDLC variants. Table 2.4 shows HDLC framing that applies to most of the variants. There are three types of formats in an HDLC frame:

- Information format for carrying user data.
- Supervisory format for control functions.
- Unnumbered format for control and management.

Table 2.4: General HDLC Framing

Field	**Flag**	**Address**	**Control**	**Information**	**FCS**	**Flag**
Field Value or Purpose	7E	Variable	See Tables 2.5, 2.6, or 2.7	Payload Data	Frame Check Sequence	7E
Field Length	8 bits	8 bits	8 or 16 bits	Variable	8 or 16 bits	8 bits

The primary distinguishing factor for each type of frame is the control field. The control field has different values for each type of framing, which is determined by the first 2 bits in the control field. In the 8-bit control format, bit 5 is the poll/final bit. It is treated as the poll bit if it is part of a frame sent by the primary station, and it is called the final bit if it is encountered in a frame sent by the secondary station. Although the HDLC variants are similar, the specific use of the control field is somewhat different for each type of HDLC. For example, Section 2.9 on PPP shows a specific variant of HDLC called LAPB. The control field can be either 8 or 16 bits in length. Tables 2.5, 2.6, and 2.7 show the format of the Control field for each type of frame.

Table 2.5 shows the information format. This format is used to transmit user data. The “I” type frames or information frames are sequenced or numbered frames for providing a connection-

oriented service. The transmitted frames include sequence numbers, and the response frames include expected sequence numbers.

Table 2.5: Information Format

1	2	3	4	5	6	7	8
0	n(s)			p/f	n(r)		

Table 2.6 shows the supervisory format. This format is for control functions only, primarily acknowledgment and requests for retransmissions. Outgoing information frames are not sequenced, but the n(r) field might contain the number of an acknowledged frame.

Table 2.6: Supervisory Format

0	1	2	3	4	5	6	7
1	0	sc		p/f	n(r)		

The format for commands used with the supervisory format is shown in Table 2.7.

Table 2.7: Supervisory Format Commands

Command	**sc Field Value**	**Name**
RR	0	Receive ready
REJ	1	Reject
RNR	2	Receive not ready
SREJ	3	Selective reject

Table 2.8 shows the unnumbered format of HDLC. This format is used for control functions and management functions such as link establishment. Table 2.9 shows some key unnumbered format commands.

Table 2.8: Unnumbered Format

0	1	2	3	4	5	6	7
1	1	un		p/f	un continued		

Table 2.9: Unnumbered Commands

Command	**Response**	**un Field Value**	**Command Name**	**Response Name**	**Purpose**
UI	UI	0	Unnumbered information	Unnumbered information	Information field contains datagram for connectionless service
SNRM		1	Set normal response mode		

Table 2.9: Unnumbered Commands

Command	Response	un Field Value	Command Name	Response Name	Purpose
DISC	RD	2	Disconnect	Request Disconnect	
UP		4	Unnumbered poll		
	UA	6	Unnumbered acknowledge		
	FRMR	11		Frame reject	
SARM	DM	18	Set asynchronous response mode	Disconnect mode	Also used by secondary station to say it is offline
RSET		19	Reset		n(s) and n(r) are reset
SABM		1c	Set asynchronous balanced mode		Set mode where sta tions are equals

2.8 Sliding Windows

One of the important functions of data link protocols is to provide a connection-oriented or reliable transmission service. A reliable service guarantees ordered packet delivery by doing retransmission of lost or damaged packets. Elsewhere in this chapter, we discussed several data communication protocols. Some of these protocols are used to carry IP datagrams, and if so, they would be interfaced to IP at the link layer. However, it is important to note that TCP/IP does not require connection-oriented service at the link layer. The TCP transport protocol actually includes its own version of sliding windows, which in theory works almost identically to the protocol described in this section. However, later we will see that the TCP state machine is far more complex. In addition, we will see that TCP has had many enhancements such as slow start to reduce congestion in busy networks.

Sliding windows provides a means of recovering from lost, erroneous, or out-of-sequence frames. Earlier protocols had to use a system called start-and-stop acknowledgment to manage the integrity of the connection. This system wasted bandwidth because the sending station had to wait for an acknowledgment before sending another frame. In contrast, sliding windows provides a better mechanism for acknowledging transmissions. The receiving station sends an accumulative acknowledgment. This is more efficient because a response from the receiving station can acknowledge multiple packets from the sending station, and the sending station need not wait for the ACK frame before sending more frames. When used in conjunction with packet queuing, this method can provide an excellent mechanism for flow control. Most modern

communication protocols use a method that is in some way fundamentally derived from sliding windows.

The sliding windows method uses a window size determined by the size of the sequence number n(s) and the acknowledgment number n(r) fields of the particular HDLC variant. The window size becomes the number of unacknowledged transmit frames that can be sent before receiving an ACK frame. Originally, HDLC reserved 3 bits for the sequence and acknowledgment fields in the information format frame and supervisory format frames, which allows for a window size of 7. Later variants of HDLC increased this field width, subsequently increasing the window size. Figure 2.4 demonstrates a typical sequence for HDLC sliding windows. In each transmitted frame, the n(s) field is set to a consecutive sequence number. The sending station responds by sending acknowledgment frames with n(r) set to the next expected value of n(s) in a received frame.

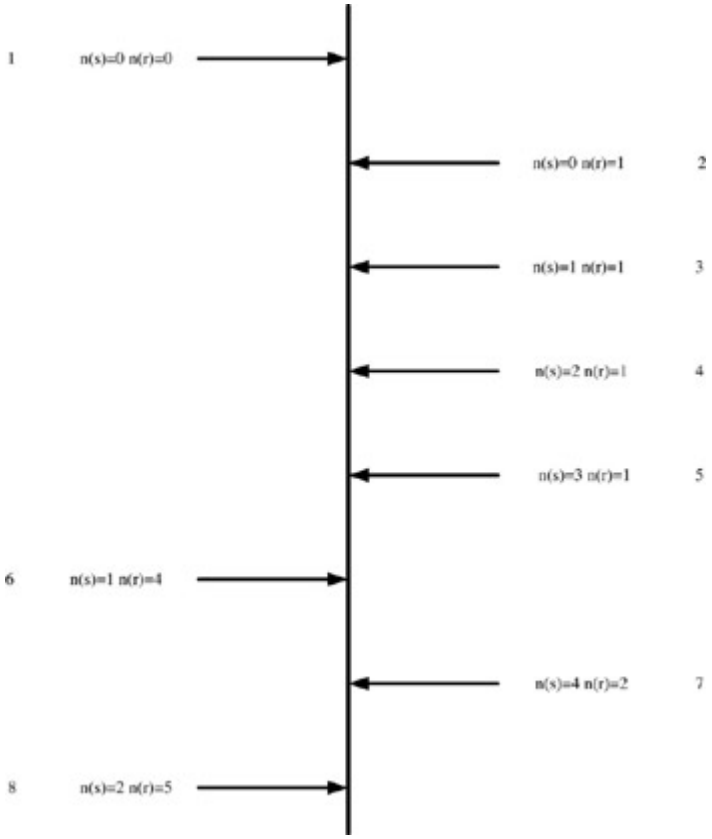

Figure 2.4: Sliding windows.

2.9 HDLC as Used in Point-to-Point Protocol

The point-to-point protocol (PPP) [RFC 1661] is widely used for providing connection for carrying TCP/IP traffic over serial links, modems, and other types of transmission links. It is perceived as a communications protocol for dial-up connections, but is also widely used for tunneled connections over Ethernet in the form of PPP over Ethernet (PPPoE) and other protocols. In this section, we are interested in how it incorporates the HDLC protocol. When PPP is carried over serial links, HDLC is generally used as the framing layer [RFC 1663]. It uses a version of HDLC called Link Access Protocol Balanced (LAPB) [RFC 1662]. LAPB changes the basic HDLC protocol shown in earlier sections by increasing the window size by increasing the size of the n(s) and n(r) fields. This effectively reduces the overhead for acknowledges because the window size governs how many packets can be transmitted before receiving an ACK.

In addition, PPP changes the LAPB protocol to allow either station to demand an immediate response from the peer by setting the p bit to 1 in the p/f field. Remember that the p/f field has two meanings, one for a command packet and a separate one for the response packet. In a command packet, it is called the *p* bit, and in the response packet, it is called an *f* bit. A response packet with the p/f field set to 1 clears the p bit state. For an example, consider the following sequence. If a receiving station detects an error in a received frame, it can send a REJ supervisory frame to the peer station with the n(r) field set to the sequence number of the frame with the error and set the p bit to 1. Since sliding windows includes implicit rejection, the n(r) value actually tells the sender that all frames previously received with a value higher n(s) value than the n(r) value are rejected. The sending station can then resend the rejected frames.

2.10 Frame Relay Bearer Service

As discussed in Section 2.6, X.25 has a performance disadvantage in that it has multiple layers with error correction. The Frame Relay Bearer Service (FR) eventually replaced X.25 for most data traffic over public networks. This was because X.25 was intended for low-quality physical links where its redundant error checking and correction at both Layers 2 and 3 is appropriate. Frame Relay eliminates the extra layer of error correction; it is intended for PHYs with far greater intrinsic reliability than the early copper-based transmission circuits that were predominant when X.25 was first used. The elimination of extra error correction makes FR more suitable than X.25 for high data rates such as those found in fast optical links up to OC-3. It is important to point out that TCP/IP traffic does not require a reliable link layer. The TCP/IP protocol has its own error checking and correction as part of the TCP protocol at the transport layer.

The PHY for Frame Relay includes support for DS0, DS1, DS3, E1, or E3. (These rate classes are shown in Chapter 1.) FR's framing layer uses Link Access Protocol Frame Relay (LAPF). The framing is similar to the HDLC framing used in X.25 but is simpler. This is because FR does not provide the error correction that is part of the X.25 packet layer protocol. See Figure 2.5 for the framing details for all three types of frames used in FR. As in HDLC framing, the FR frame is preceded by a flag consisting of a 0 bit followed by five consecutive 1 bits, which is a value of 0x7e. This flag is used to indicate the frame start. The frame is also terminated with a similar flag having the same value of 0x7e. The reason for 0x7e is that it used to differentiate control data,

payload data, and detected errors similar to the bit stuffing technique used in HDLC. Refer to Section 2.5.4 for more details about basic HDLC and bit stuffing. Table 2.10 shows the frame format for FR.

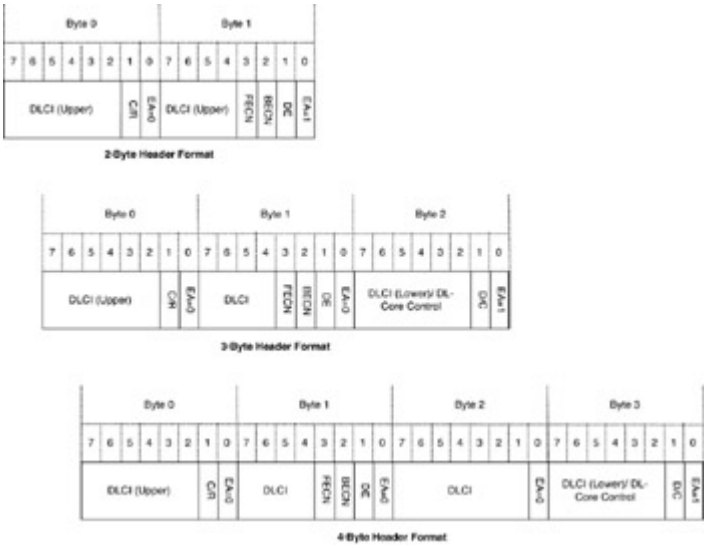

Figure 2.5: Frame Relay header formats.

Table 2.10: Frame for Frame Relay Bearer Service

Flag	**Header**	**User Data**	**Frame Check Sequence**	**Flag**
0x7e	Table n	Payload Data	FCS	0x7e
1 byte	2, 3, or 4 bytes	Variable	2 or 4 bytes	1 byte

FR allows multiple virtual circuits to be multiplexed across the same physical link. The virtual circuits are differentiated by the FR header field. which contains a field known as Data Link Connection Indicator (DLCI). This field is used by FR routers to determine how to send a frame to its destination. There are four different DLCI lengths: 10, 16, 17, or 23 bits. The values of the D/C and EA bits determine the type of DLCI field. Because of the varying DLCI field, there are three different header formats for HR: 2 byte, 3 byte, and 4 byte. The multiplexing techniques make use of a field in the header called the Data Link Connection Indicator (DLCI). The function of the DLCI field is to be a virtual circuit identifier. This is similar to the address field in a LAN network packet. FR packets are switched among channels by the FR routers based on the value of the DLCI field. Specific descriptions of each of the fields in the Frame Relay header format are shown in Table 2.11.

Table 2.11: Explanation of Fields in Frame Relay Header

DLCI	**Data Link Connection Indicator**	**The identifier of the SVC or PVC to which this frame belongs.**
C/R	Command or	Not used by the Core Frame Relay Protocol.
EA	Extended Response bit Address	Allows for multiple DLCI or address lengths. Zero indicates that another header byte is to follow. One means that this header byte is the last.

Table 2.11: Explanation of Fields in Frame Relay Header

DLCI	**Data Link Connection Indicator**	**The identifier of the SVC or PVC to which this frame belongs.**
FECN	Forward Explicit Congestion Notification	Used for congestion avoidance and traffic control.
BECN	Backward Explicit Congestion Notification	Used for congestion avoidance and traffic control.
DE	Discard Eligible	Discardable packets are marked with this bit if congestion exceeds a threshold.
D/C	DLCI or DL-Core Indicator	One indicates that DLCI bits in the last byte are used for DL-Core functions. Zero means that DLCI bits in the lowest byte extend the DLCI address space.

Frame Relay includes a signaling protocol for managing SVCs, but FR service can be provided over PVCs as well. The frames that actually do the negotiation to control the SVCs don't travel in the same link as the frames carrying data traffic. This signaling method is called *out-of-band signaling* when the signaling frames are transmitted over a special channel. In addition, signaling frames use a different form of the header than the header format for data frames. The signaling protocol used with FR is ITU standard Q.931.

As discussed earlier, the mechanism for reliable service used in X.25 was based on HDLC and its sliding windows protocol. FR does not provide reliable or connection-oriented service or a specific mechanism for flow control at Layer 2; instead, it provides a mechanism called *traffic engineering*. FR establishes a Committed Information Rate (CIR) for each channel based on the nominal amount of available bandwidth. The traffic engineering is provided by the use of three fields in the CR frame header. Frames with the DE bit set to 1 can be discarded if the transmission rate during a burst period exceeds the CIR. The Forward Explicit Congestion Notification (FECN) field indicates to the sender that there is congestion on the network. Likewise, the Backward Explicit Congestion Notification (BECN) field indicates to a receiving station that there is congestion on the network. A node on the network can start flow control if it detects the FECN or BECN bits in a frame. The flow can be maintained until adjacent nodes have time to clear out the congestion.

2.11 Asynchronous Transmission Mode (ATM)

ATM is one of the most recent of the broadband or WAN protocols. We should point out that Packet over Sonet (PoS) is more recent, scales to the highest optical speeds, and has a well-defined interface for IP datagrams. However, we will confine our discussion to ATM in this chapter because it is interesting in that it attempts to provide all kinds of imaginable service, data rates, and traffic classes. ATM has similarities to the Frame Relay Bearer Service or even the earlier X.25. ATM is a complex protocol, so we will not be able to discuss it in detail here. It provides the basic infrastructure for the modern PSTN. In the 1990s, it was thought that it would replace TCP/IP for most data transmission because of its capability to support Quality of Service (QoS). However, with the amount of dark fiber and widely available bandwidth, TCP/IP is very capable of providing sufficient bandwidth for voice and video applications. Moreover, as we will

see in Chapter 9, "The Network Layer, IP," Linux IP can be configured to do routing based on traffic classes. ATM interfaces with TCP/IP where the enterprise network meets the PSTN. ATM is too involved to be covered in this book in detail. Refer to the bibliography for more detailed information on the history and evolution of this complex protocol suite. However, it is important to point out a few facts from this history. In the 1980s, the telephone companies were looking for a way to deliver differentiated digital services to customers using the PSTN. They came up with a suite of protocols called ISDN that specified how a user would interface to the PSTN to access various services with different capacities and degrees of reliability. ATM was originally called Broadband ISDN (B-ISDN) to specify available services on bit rates higher than what was available with ISDN. ATM is entirely scalable from a single DS0 all the way to OC192. ATM defines a complete complex protocol stack, and can provide telephony-type voice circuits as well as connectionless data services. It can provide different needs for quality of data transmission (QoS).

ATM is designed to deal with transmitting data, voice, and video simultaneously. Instead of being based on frames or packets, it is implemented with a small Protocol Data Unit (PDU) called a *cell*. The ATM cell is 53 bytes and includes a 5-byte header and a 48-byte payload. This particular cell size is used because it is the optimum size for individual submodulation of the carriers used with the digital data hierarchy found in all the public networks of the world, North America, Europe, and Asia. The small cell size allows for the interrupted transmission of individual voice packets, but is also scalable to far higher data rates. Because of the small cell size and high speeds, header manipulation and cell switching is done in hardware or firmware instead of in slower software. However, because of the small cell size and 5-byte header, it has more than 10% overhead.

2.11.1 The ATM Stack

The ATM stack is far more complex than TCP/IP. It defines both User-to-Network Interfaces (UNI) and Network-to-Network Interfaces (NNI). ATM includes a signaling protocol for negotiating connections. It provides a management interface to allow for provisioning of PVCs and gathering of cell-level statistics. The UNI specification includes definitions of classes of service and how to specify service types when interfacing to an ATM network. The different service classifications are provided for video, voice, and data.

Because of the complexity of ATM, it is shown in a layered model that is similar to OSI but is three dimensional, including multiple planes for data, management, and control. ATM is interfaced to IP through ATM Adaption Layer 5 (AAL5) with the use of a Segmentation and Reassembly (SAR) module to build larger IP packets from the smaller ATM cells when they are received at the interface and to break up the IP packets into cells when they are transmitted. Refer to Figure 2.6 for a picture of the ATM stack showing how it interfaces to IP.

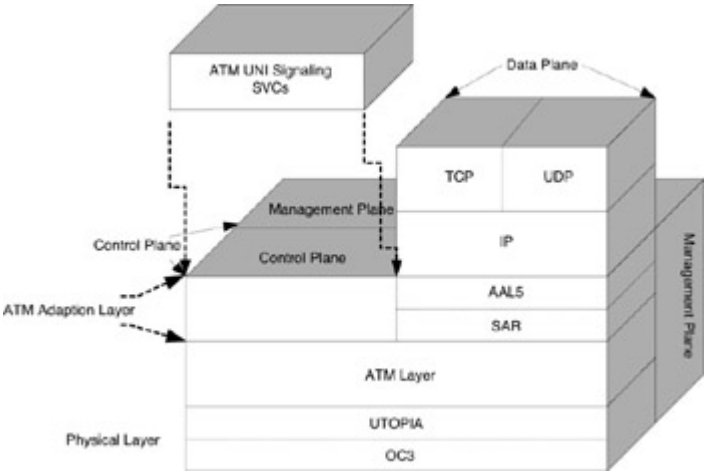

Figure 2.6: ATM stack.

The lowest layer in the ATM stack is the PHY. It is divided into upper and lower sublayers. The lower sublayer handles the physical transmission details for the optical or electrical interface and generally consists of an interface to the Universal Test and Operations PHY Interface (UTOPIA) bus, which is a parallel interface that hides the implementation details of the TDM multiplexing from the PHY layer of ATM. The upper sublayer in the PHY is the Transmission Convergence (TC) sublayer, and there are TCs defined for each of the supported PHYs, DS1, DS3, up to OC3. The TC layer maps individual ATM cells onto a TDM data stream.

2.11.2 ATM Network Interfaces

The next higher layer in the ATM stack is the ATM layer. This layer is responsible for differentiating among the different types of cells during routing and switching. There are two types of ATM layers: one implements the Network-to-Network Interface (NNI), and the other implements the User-to-Network Interface (UNI). The NNI ATM layer is actually a switch that moves individual ATM cells from one or more input data stream ingress points to one or more output data stream egress points. The NNI ATM switch is also known as an *Intermediate System* (IS); it controls the circuit connections from ingress point to egress point. In the IS, cells are switched on an individual level because cells are individually multiplexed among different virtual channels depending on the type of channels and cells. The switching needs to be very fast to handle the high data rates and therefore is usually implemented in hardware called an ATM *switching fabric*. The ATM switches or ISs include the management plane but they don't include the AAL layer.

ATM is connection oriented; therefore, before any cells can be transmitted, an endpoint-to-endpoint connection must be established. As in other protocols discussed earlier, each connection becomes a VC. Each VC is full duplex or bi directional. The circuits are built by the ISs in the network by building two end-to-end chains of connections. A connection for each direction is built from Virtual Channel Links (VCL) and Virtual Path Links (VPL). See Figure 2.7 for a picture of the ATM UNI cell format. Figure 2.8 shows the ATM NNI cell format.

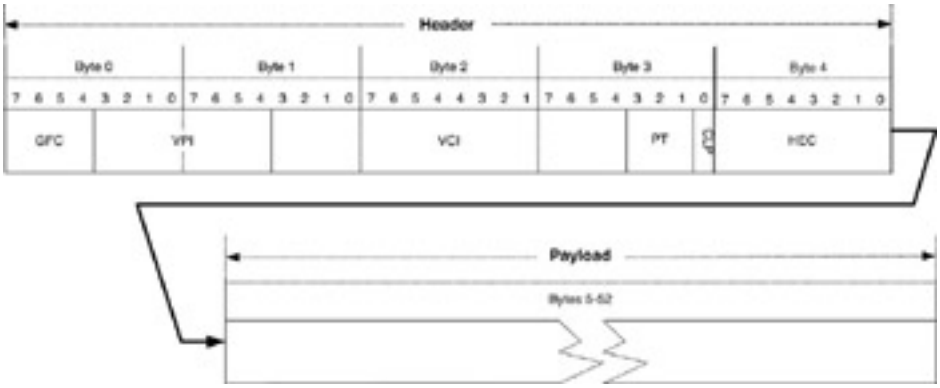

Figure 2.7: ATM UNI.

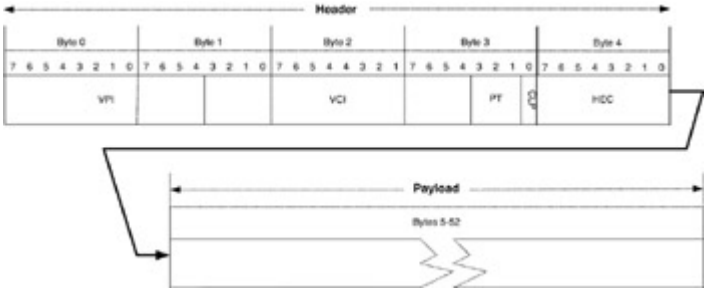

Figure 2.8: ATM NNI.

2.11.3 ATM UNI Service Class Definitions

The UNI, also known as an End System (ES), includes the ATM Adaption Layer (AAL). The ES also has an ATM layer but it is simpler than the ATM layer in the IS. AAL is defined as five sublayers: AAL1 through AAL5. The AAL is further sub divided into a Service Specific Convergence Function (SSCF) and Service Specific Convergence Sublayers (SSCS). The Segmentation and Re-assembly (SAR) firmware looks at the cell headers and subdivides the cells by class and passes them to the ATM stack through a specific SSCS, depending on the service class and which AAL level interface is required for each service class. Refer to Table 2.12 for a summary of the ATM UNI traffic class definitions.

Table 2.12: ATM UNI Traffic Classes

Class	Sender—Receiver synchronization	Bit Rate	Connection	AAL	Content Type
A	Yes	CBR	CO	1	Voice circuits
B	Yes	Rt-VBR	CO	2	Time critical packets, voice signaling, video
C	No	VBR	CL	3, 4, 5	Packet data, FR
D	No	VBR	CL	5	IPoA
					VBR: Variable Bit Rate
					CBR: Constant Bit Rate
					CO: Connection Oriented
					CL: Connectionless

Table 2.12: ATM UNI Traffic Classes

Class	Sender—Receiver synchronization	Bit Rate	Connection	AAL	Content Type
					FR: Frame Relay
					IPoA: Classical IP over ATM
					Rt-VBR: Real-time VBR

Four classes are defined to carry each of the traffic types carried in ATM cells. In addition to the traffic classes, there are two types of bit rates and two connection types. Each of the five sublayers corresponds to different service classes and traffic types to provide constant or variable bit rate and streaming or bursty traffic. Refer to Table 2.12 for a list of the ATM UNI traffic types. Like Frame Relay, ATM contains mechanisms for congestion control based on a concept called Available Bit Rate (ABR). ABR is defined for traffic engineering and is not explicitly defined as part of the UNI service definition.

As can be seen, the ATM protocol is very complex. Some IP-based data such as Voice over IP (VoIP) and other applications demanding specific QoS must interface to the ATM stack in more complex ways. However, for most pure data applications, the actual interface from IP to ATM is fairly simple conceptually. It consists of the AAL5 layer and the SAR. The internal implementation of the SAR is complicated, but in most applications it is implemented in firmware or hardware as part of a PHY chip.

2.12 Summary

In this chapter, we discussed the differences between LAN and WAN or broadband protocols. We covered some significant broadband protocols in more detail. We saw how these protocols evolved. We covered X.25, HDLC, and Frame Relay in some detail. We covered ATM in more detail and showed how it is the most complex of the family of broadband protocols. In Chapter 3, we will introduce the basics of TCP/IP. We will cover some implementation details of the link layer, Layer 2 on the OSI model, and show how attention paid to Layer 2 will help us understand how TCP/IP can be made to interface to some of the protocols covered in this chapter.

Chapter 3: TCP/IP in Embedded Systems

TCP/IP is the dominant protocol suite in modern networking. However, TCP/IP is far from new technology; its origins go back to the 1970s. The popularity of Linux is the result of an evolutionary process that dates back around the same length of time. The third trend is the growth in embedded systems. These two trends have merged, so now Linux is one of the most popular operating systems (OSs) to be deployed in embedded systems today. A major advantage of Linux is the mature and stable implementation of its TCP/IP stack and the fact that the Linux implementation has proved to be one of the most successful platforms for modern networked devices. This chapter builds on Chapters 1 and 2 by exploring the TCP/IP protocol stack and discussing some aspects of the TCP/IP of particular importance to embedded systems engineers and other people who care about the internal implementation.

3.1 Introduction

To understand the events that led up to the current popularity and near dominance of the Linux implementation of TCP/IP, it is important to divert our attention a little to the history of the protocol suite, the open source movement, and the Linux operating system. TCP/IP networking goes back to earlier days of government-funded computer-related research in the 1970s. Academic and research institutions scattered across the United States were using computers that needed to talk to each other, and researchers looked into new open protocols to exchange generic information. This research led to the development of the first protocols in the TCP/IP suite. At the time of development of TCP/IP, the Unix system was in wide use as an alternative to proprietary operating systems in common use. These proprietary OSs were only available from computer manufacturers such as DEC and IBM and ran only on the respective vendor's hardware architecture. The first version of Unix was developed by AT&T Bell Labs as a platform for developments in telephony. In the late 1970s and early 1980s, the University of Berkeley Computer Systems Research Group (CSRG) began distributing a version of Unix available for DEC hardware called BSD (Berkeley Software Distribution).

During this time, ARPANET, the forerunner of the TCP/IP protocol suite, was already in use at United States Defense Advanced Research Projects Agency (DARPA) funded research labs. DARPA contracted with BBN to have TCP/IP integrated into the BSD OS as part of a government contract. Since the BSD OS was already being distributed to universities in source form, TCP/IP suddenly became available in source form to students, researchers, and engineers. It quickly became an accepted industry standard and was ported into all sorts of small and embedded systems running proprietary OSs. In the meantime, various Unix derivatives were in use in embedded systems, and these OSs were the first ones to introduce TCP/IP to the embedded systems engineer.

3.2 A Note on TCP/IP Implementation

It is important to draw a distinction between the networking protocol and its implementation. The networking protocol is a specification where a sequence of exchanges between computers is specified in detail. Let's look at a simple example without specifying any particular OS. In Figure 3.1, machine A broadcasts a packet requesting the address of machine B in order to send a

packet directly to B. A server on the network, Q responds to the request from A with a packet containing the address of B. A then sends a packet directly to B.

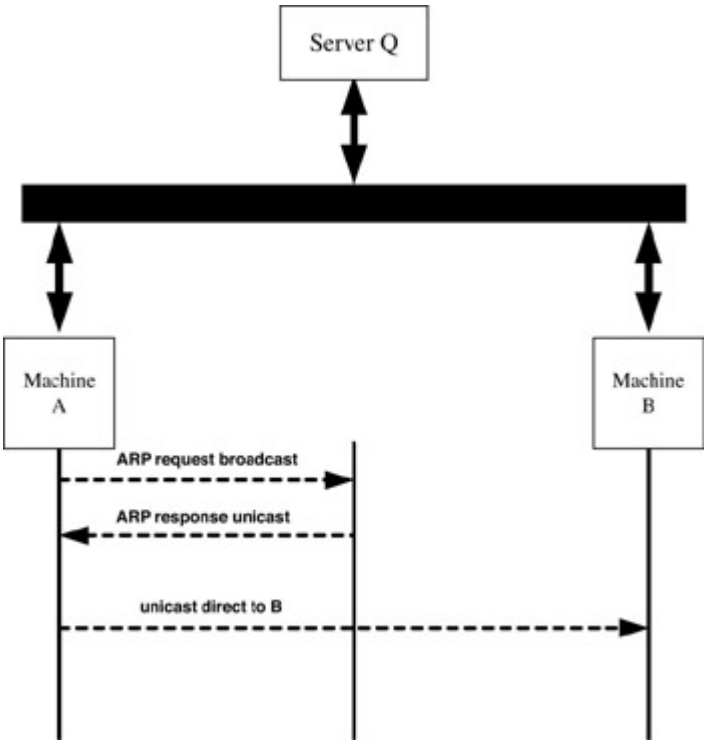

Figure 3.1: Simple protocol example.

An implementation is how the software performs the protocol functions internally in a given computer and OS. For example, consider the steps that are executed internally in the TCP/IP stack to implement the generic example protocol in Figure 3.1. An application constructs a buffer for a packet to send. The buffer is passed to the User Datagram Protocol (UDP) input routine, which copies it into an internal buffer and then passes it to IP. The buffer is queued up at the Ethernet network interface when the IP output routine calls the driver's send function. The driver makes a call to the address resolution function to get the destination MAC address. The address resolution function checks the ARP cache to see if there is already an entry for the destination IP address. In this example, it doesn't find an entry, so it broadcasts an ARP request packet. When the response to this packet is received, the ARP cache is updated and the address resolve function returns with the destination MAC address. The driver then takes the MAC address, puts it in the packet MAC header, and transmits the packet.

3.3 TCP/IP in Terms of the OSI Model

In this book, as in most publications about networking, the TCP/IP protocols or "stack" are described in terms of the OSI (Open System Interconnect) reference model. Refer to Chapter 1, "Introduction," for more about networking fundamentals and the OSI model. In this chapter, however, we will see how the OSI model fits in with TCP/IP. The OSI model was specified in the 1970s by the International Standards Organization (ISO) to promote interoperability between widely diverse computer architectures and OSs. The reference model describes a seven-layer architecture. At the time the OSI model was specified, TCP/IP was already established, so not all

the components of TCP/IP fit neatly into the OSI layered definition. However, the OSI model remains the best framework to explain network protocols in general and the TCP/IP stack in particular. After the original Berkeley version of TCP/IP was distributed and widely used, it came to be discussed in terms of the OSI model. The TCP/IP stack in terms of the model is shown in Figure 3.2.

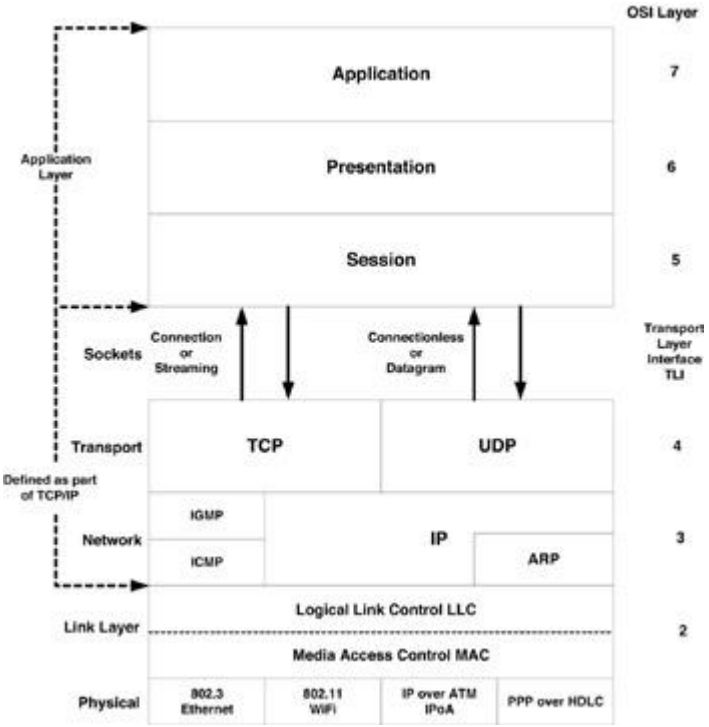

Figure 3.2: TCP/IP and the OSI seven-layer model.

In modern times, we have essentially two architectures, *little endian* and *big endian*, where both use 8-bit bytes and 4-byte words. However, in ancient times when the OSI model was specified, things were more complicated. It was like the Wild West in that there was no order, no coordination, and a jumble of incompatible computers. Different manufacturers of computers used different byte and word lengths, as well as different bit and byte ordering. A well-defined method to implement computer protocols was sorely needed.

Two guiding principles allow protocol stacks to be implemented as shown in the OSI model: *information hiding* and *encapsulation*. Each layer hides its implementation details from the adjacent layers above and below. In addition, as two machines communicate, each layer in the transmitting machine has a peer-to-peer relationship with the same layer in the receiving machine. Encapsulation is the primary mechanism that allows the layers to hide information from the layers above and below. As the data being readied for transmission travels down the stack, each layer puts its header in front of the data it receives from the layer above. In the receiving machine, each layer strips its header off after receiving the data from the layer below.

3.4 Physical Layer

The physical layer (PHY) is responsible for the modulation and electrical details of data transmission. An example of a PHY used in local area networks (LANs) is Ethernet. Ethernet uses a technique called Carrier Sense Multiple Access with Collision Detect (CSMA/CD). Another example of a PHY is a *T1* interface, which is used in Wide Area Networks (WANs). Refer to Chapter 2, "Broadband Networking Protocols of Yesterday and Today," for more information about WANs. Each machine has a unique physical address that is usually called a Media Access Control address (MAC) on LANs or a station ID on WANs. The MACs can support several addressing modes, including *unicast*, *multicast*, and *broadcast*. Unicast is the transmission of a frame to a specific single recipient; multicast is the transmission of a frame to a specific group of recipients; and broadcast is the sending of a frame to all the machines that can be reached in a subset of the network topology.

3.5 Data Link Layer

The data link layer isolates the network layer above from the electrical transmission details in the layer below. One of the responsibilities of the data link layer is to provide an error-free transmission channel known as Connection Oriented (CO) service. Generally, CO service at the data link layer is not required because TCP/IP assumes that the lower layers do not have any type of error recovery. If a user requires reliable transmission, she should use SOCK_STREAM type sockets, which transmit and receive data using the TCP protocol. In the TCP/IP suite, TCP provides reliable service at the transport layer. However, in IP-based networks, there are some cases where CO or reliable delivery service is used at the data link layer. When we use the CO service at the data link layer, the network interface driver does not consist of hardware support. Instead, the driver interfaces IP to another complete protocol stack. The best example is Point-to-Point Protocol (PPP). PPP presents a common example of a reliable link because it contains several layers of control protocols that must negotiate a reliable connection before it is ready to carry IP packets. In examples such as PPP where IP runs over a reliable link, often the purpose of the link is to forward IP across a public carrier-based network of some sort. Examples of public carrier networks are X.25 or ATM.

In some TCP/IP stack implementations, there is a filtering layer where incoming packets or frames can be intercepted before being passed up to IP. This filtering occurs at Layer 2 in terms of the OSI model. An example of filtering would be Linux netfilter. For more details about netfilter, refer to Chapter 4, "Linux Networking Interfaces and Device Drivers." Actually, the simplest form of filtering occurs automatically at the PHY. The link layer sets up the PHY to receive only packets with its own MAC address in the destination field along with multicast packets and broadcast packets. (An exception is made when the PHY is in promiscuous mode, discussed later in this chapter.) Another function of the link layer is to establish the type of framing to be used when the IP packet is transmitted. For example, if the incoming packet is from an Ethernet interface, it can have two types of framing. One is often called Ethernet type II framing; the other, used for Ethernet, is 802.3 type framing. The length field in the 802.3 packet is at the same displacement as the type field in the Ethernet type II frame. See Figure 3.3 for an illustration of Ethernet Type II framing, and Figure 3.4 for an illustration of 802.3 type framing. The 802.3 type framing also has a type field that contains the protocol number carried by the

frame. Table 3.1 shows the protocol numbers commonly used for IPv4 networking. See Section 3.11 for more information about Internet standards and protocol numbers.

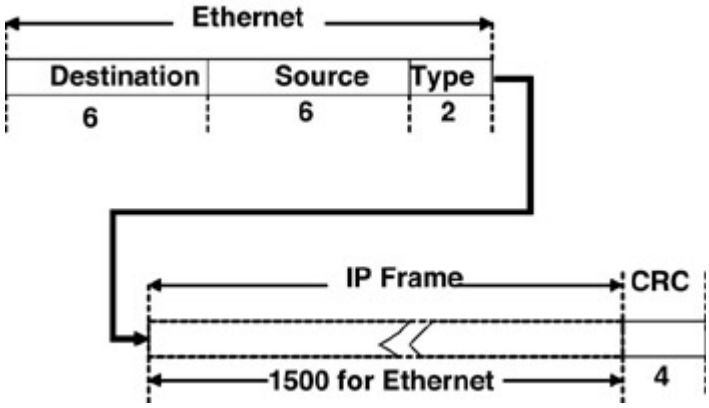

Figure 3.3: Ethernet framing.

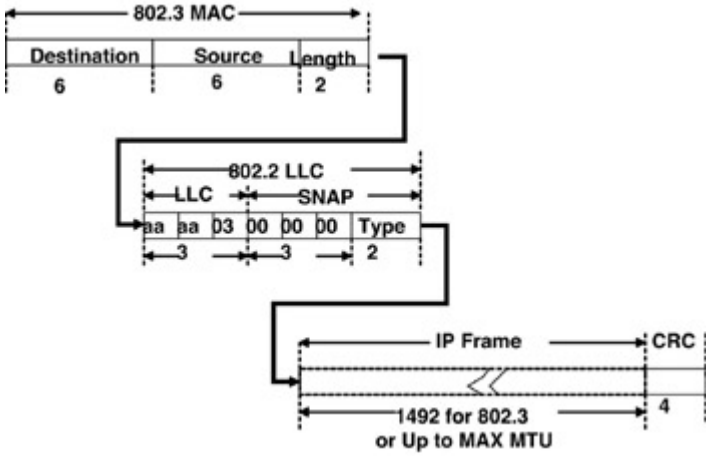

Figure 3.4: 802.3 framing.

Table 3.1: Protocol Types for Type Field in Ethernet MAC Header

0x0800	IP
0x0806	ARP
0x8035	RARP

3.6 Layer 3, the Network Layer—IP

The network layer in the TCP/IP protocol suite is called IP, for Internet Protocol [RFC 791]. This layer contains the knowledge of network topology. It includes the routing protocols and understands the network addressing scheme. Although the main responsibility of this layer is routing of packets, it also provides fragmentation to break large packets into smaller pieces so they can be transmitted across an interface that has a small Maximum Transmission Unit (MTU). Another function of IP is the capability to multiplex incoming packets destined for each of the transport protocols. See Figure 3.5 for an illustration of the IP header.

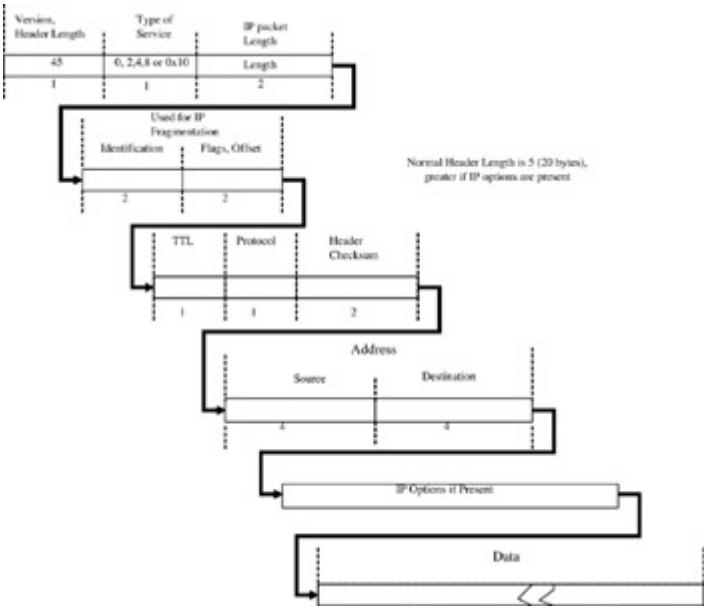

Figure 3.5: IP header.

3.6.1 IP Addressing

This book does not attempt to duplicate the work of many excellent books available about TCP/IP network configuration and administration. Instead, we concentrate on the internal implementation of Linux TCP/IP. Before diving into IP routing in later chapters, it is necessary to introduce the IP addressing scheme. Differentiating among the classes of addresses is a significant function of the IP layer. There are three fundamental types of IP addresses: unicast addresses for sending a packet to an individual destination, multicast addresses for sending data to multiple destinations, and broadcast addresses for sending packets to everyone within reach.

It is conventional to express IP addresses in dotted decimal notation where each byte of the address is written as a decimal number. The IP addresses are divided into classes. The first three bits of the IP address determine the class [RFC796]. The class of address determines the size of the network; it defines what portion of the address is the network portion and what portion is the host portion. The network portion is the higher-order side of the address, and the host portion is the right part of the address. See Table 3.2 for a layout of the IP addressing scheme. Class A and B addresses were once used widely in an enterprise for all the hosts on the network. After awhile, as the popularity of the Internet increased, A and B addresses became less often used because they allow too many hosts on a network. Routers became more prevalent so an enterprise could implement multiple smaller networks at lower cost than what was once possible. Therefore, class C addresses became more common; however, the problem with class C addresses is that they don't have enough hosts.

Table 3.2: IP Addressing

Class	Lower Bound of Network	Upper Bound of Network	Lower Bound of Host	Upper Bound of Host
Default route is	0.0.0.0	0.0.0.0	0.0.0.0	0.0.0.0

Table 3.2: IP Addressing

Class	Lower Bound of Network	Upper Bound of Network	Lower Bound of Host	Upper Bound of Host
all zeros				
A—8 bits net; 24 bits host	0.0.0.0	126.0.0.0	126.0.0.0	126.255.255.255
Loopback	127.0.0.0	127.0.0.0	127.0.0.0	127.x.x.x
B—16 bits net; 16 bits host	128.0.0.0	191.255.0.0	128.0.0.0	128.0.255.255
C—24 bits net; 8 bits host	192.0.0.0	223.255.255.0	192.0.0.0	192.0.0.255
Multicast	224.0.0.0			239.255.255.255

In modern times, available network address space on the Internet is extremely limited. An organization is typically assigned only one or very few addresses, and all the traffic destined to an organization is directed to addresses within this group. This is called Classless Interdomain Routing (CIDR) [RFC 1519]. Internally, organizations can translate incoming and outgoing packets from the external address to an internal address space. The internal network addresses are subdivided by using subnets [RFC917].

The netmask can be used to separate the network portion of the address from the host portion, but the dividing point is not necessarily on one of the class boundaries shown in Table 3.2. This is called *subnetting*. Netmasks are usually shown as hex numbers, particularly in systems that have a Unix heritage such as Linux. The subnet mask subdivides the host portion of the address into a subnet identifier and a host identifier. For example, a network ID might be 191.254.0.0 and the subnet could be 0xffffff80. This would give 128 host IDs within the network identified by 191.254.0.0.

3.6.2 Address Resolution Protocol

The Address Resolution Protocol (ARP) is sometimes thought of as being in the link layer, but for the purposes of this explanation, it will be placed in the network layer alongside IP. The purpose of ARP is to determine what the physical destination address should be that corresponds to the destination IP address. As we will see later in Chapter 4, the IP layer hands over its packet to the interface driver when the IP portion of the header is complete, but IP has no way of knowing the physical transmission details needed to determine the hardware address. However, the driver or network interface knows how to form the hardware or MAC address but doesn't know the address associated with the specific destination for this packet. The driver checks the destination to see if the address is known. If not, the resolve function in the ARP module finds out the destination address. If the ARP cache contains an entry for the physical address, the address resolve function can be retrieved immediately. Otherwise, the ARP module sends out a

broadcast packet to locally connected hosts called a *who-has* to see if anybody knows the MAC address associated with this IP address. When a reply to the who-has is received, an entry is created in the ARP cache. Later, the network interface driver can access the address from the local ARP cache. As we shall see in later chapters, Linux manages the ARP cache in a somewhat more complex way. The ARP cache is created from the generic destination cache. A pointer to the destination cache entry is placed in the packet before it is handed off for transmission. Later when the packet is being transmitted, the MAC header can be retrieved quickly from the cache if the address has been resolved. If not, the packet is passed to ARP where it is queued while ARP gets the destination address.

3.7 Layer 4, the Transport Layer

The transport layer in TCP/IP consists of two major protocols. It contains a connection-oriented service reliable service otherwise known as a *streaming service* provided by the TCP protocol. In addition, TCP/IP includes an individual packet transmission service known as an *unreliable* or *datagram service*, which is provided by UDP. Since, historically, it was the function of the data link layer to provide an end-to-end connection-oriented service that includes error checking and recovery, one wonders why there is a need for a reliable delivery method at the transport layer. The reason why TCP/IP evolved this way is that it was designed to be carried over Ethernet and other LAN protocols, which don't provide a connection- oriented service at the link layer.

3.7.1 User Datagram Protocol

UDP is used where there is a need to send individual packets or datagrams. Each packet is sent as an individual transmission—there is no sequencing of packets and no error detection and recovery mechanism to retry transmitting of lost packets. UDP does not provide a means of ordering the packets. It is possible that packets could arrive at the destination in an order different from the one in which they were sent. Application code using UDP should make no assumptions that packets will arrive in order or arrive at all. It is necessary for applications using UDP to code in their own acknowledgment scheme if reliability is a concern. UDP provides a simple interface to the socket layer above and to the IP layer below. Figure 3.6 shows the UDP header format.

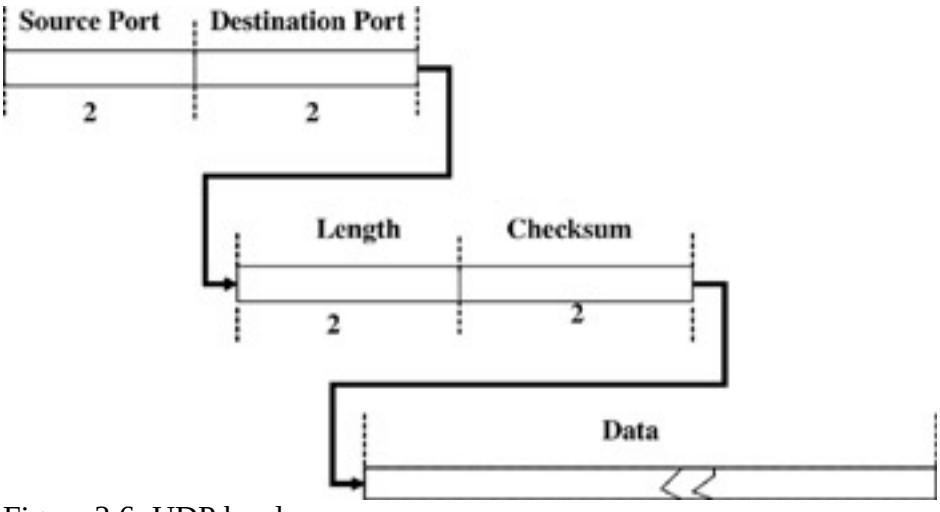

Figure 3.6: UDP header.

3.7.2 Transmission Control Protocol

TCP is used where an application requires a reliable or streaming service. Its internal implementation details are almost completely invisible to applications using TCP. To the application program, it looks like a stream of bytes. Most of the well-known and common applications used in the modern connected world use TCP such as FTP and Telnet. The most common of these is the HTTP protocol used by Web browsers. TCP guarantees that each byte will arrive in sequence, and if an intermediate packet was dropped or errors are detected, the protocol handles retransmission of dropped or erroneous packets. See Figure 3.7 to see the fields in the TCP header.

Figure 3.7: TCP header.

TCP is quite complex and is covered much more in depth in Chapter 8, "Sending the Data from the Socket through UDP and TCP," where we delve into the internals of the Linux implementation of the TCP. However, in this section, we will go over some of the fields in the header to illustrate in general terms the basic function of TCP. In Chapter 2, "Broadband Networking Protocols of Yesterday and Today," we covered the sliding windows algorithm. When using the TCP/IP protocol suite, users must use the TCP transport when they want to have a reliable stream-oriented or connection-oriented delivery. The window size, sequence number, and acknowledgment number fields are used to implement TCP's version of sliding windows. TCP divides the data stream into segments. The sequence number and acknowledgment number fields are byte pointers that keep track of the position of the segments within the data stream. The flag bits are used to maintain the state of the connection. See Figure 3.8 for more detail on the TCP header fields used to maintain the connection state.

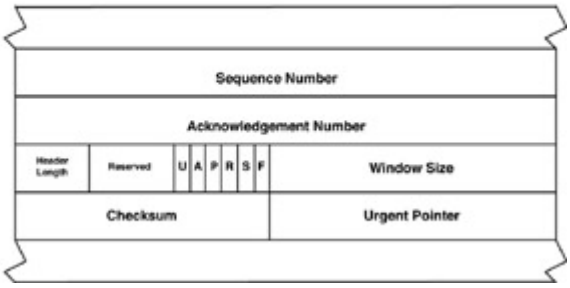

Figure 3.8: TCP segments.

3.8 Sockets—The Transport Layer Interface

The socket API is probably one of the most widely used APIs in software development. It is a very versatile and well understood interface. The socket interface supports both UDP (SOCK_DGRAM) and TCP (SOCK_STREAM). The most common socket "standard" is known as BSD sockets. Almost all socket implementations conform to the BSD sockets, and Linux is no exception. Application code that uses the socket API can be ported to Linux with virtually no changes. The socket API, fundamental to client-server programming, is the underlying basis of Web services, XML, JSP, or many other common Web-based programming paradigms. In Chapter 5, "Linux Sockets," we walk through the socket API. We show how sockets work internally and how the socket API works with TCP, UDP, and other protocols. In addition, we show how sockets can be used to access configuration items specific to the Linux operating system. For a complete and comprehensive explanation of socket programming from the applications perspective, refer to *Unix Network Programming* by W. Richard Stevens [STEV98].

A main advantage of sockets in the Unix or Linux environment is that the socket is treated as a file descriptor, and all the standard IO functions work on sockets in the same way they work on a local file. The fundamental mode of programming sockets, particularly TCP or streaming sockets, is called client-server programming. The basic difference between a client and a server is that the server listens for a connection on a socket, and the client initiates the connection to the server through a socket.

3.9 Application Layer Protocols

The OSI model discussed in Chapter 1 defines seven layers. The TCP/IP specifications really only define Layers 3 and 4. Therefore, the TCP/IP protocol suite specification does not specify anything above the transport layer. From the viewpoint of TCP/IP, the upper layer protocols can be grouped and considered application layer protocols. It is useful, though, to discuss the upper layers briefly to create a complete picture of how TCP/IP fits into the seven-layer ISO model.

3.9.1 Layer 5, Session

The session layer can be thought of as analogous to a signaling protocol where information is exchanged between end points about how to set up a session. The session layer was originally

intended for multiple users logged in to a central terminal server, what used to be called *time-sharing* in ancient times. This was a common practice with early Unix systems when most people used a computer through dumb terminals. Now that Graphical User Interfaces (GUIs) have became the normal mode of interaction with computers, the early use of session layer protocols is not as critical. One widely used session layer protocol is the Telnet protocol, which has been used for many years for virtual terminal access to remote systems [RFC853]. In the TCP/IP world, it is implemented as an application layer protocol over SOCK_STREAM (TCP) sockets. A good example of a session layer protocol in modern use is the Session Initiation Protocol (SIP) [RFC 3261]. This protocol is for controlling the creation and maintenance of sessions between two or more participants for telephone calls and multimedia exchange and distribution.

3.9.2 Layer 6, Presentation

Presentation is defined as Layer 6 of the OSI model. However, as with the session layer, from the viewpoint of TCP/IP the presentation layer is the application's responsibility. The purpose of the presentation layer is to provide a user with a view of data that is independent from either its implementation details on any particular computer architecture or details about how the data is transmitted over a network. The presentation layer generally consists of a data description language. A data description language that was originally intended for use with the OSI protocol stack is Abstract Syntax Notation (ASN.1). This language is now commonly used as the basis of the Structure of Management Information (SMI), which is the method used to represent manageable objects as part of the Simple Network Management Protocol (SNMP). Other examples of presentation layer protocols could include systems such as Common Object Request Broker Architecture (CORBA) or Extensible Markup Language (XML) because their primary interest is to provide an implementation-independent way to view varying data resources and share that data between dissimilar systems. All these protocols run as applications outside the Linux kernel.

3.9.3 Layer 7, Application

Any software that uses the Socket Transport Layer Interface API to send and receive data through TCP/IP could be considered an application layer protocol. From a theoretical perspective, most of the application layer protocols include elements that are at the session or presentation layer. Some of these protocols like Telnet were discussed previously. A protocol that is pretty much at the application layer is the most common protocol used with the Internet: the HyperText Transfer Protocol (HTTP) used in Web browsing. In addition, the File Transfer Protocol (FTP) is an application protocol that has elements of the session or presentation layers.

3.10 TCP/IP in Embedded Systems

Before TCP/IP came to be widely used in embedded systems, it was common for embedded applications to run in systems without any OS at all. Sometimes embedded engineers would use a minimal OS containing no interface to integrate a networking stack, and most embedded engineers would use in-house designed operating systems if they had any OS at all. However, as

the cost of hardware and memory decreased, computers for embedded applications needed to support software with increased complexity, so commercial embedded OS vendors moved into the market. The first commercial or outside operating systems consisted of small multitasking kernels that advertised high efficiency and low memory requirements. These OSs, however, didn't have any standard IO interfaces, and didn't support file systems, TCP/IP, or any other networking interfaces. In the meantime, embedded systems became more and more widely used and they were incorporated in just about any machine or device that is controllable, configurable, or programmable. As the complexity of embedded applications increased, there was an increasing need for more features in the operating systems such as file systems and TCP/IP networking. Commercial vendors filled the need, selling operating system software that included TCP/IP as a built-in component. These vendors collected royalties on each target system. As hardware became cheaper, the royalty became a more significant factor in the cost of the end product. Therefore, more and more embedded systems engineers are now looking for an open source solution for their embedded OS and TCP/IP networking stack.

Originally, Unix was too big and complex for most embedded applications. Now with cheaper memory and faster cheaper processors, the memory requirements of the OS is a less significant factor in the overall design. Therefore, Linux becomes increasingly practical for embedded systems. Along with Linux, embedded systems designers get a robust implementation of TCP/IP along with all the source code. In addition, as the popularity of Linux increases, it gets used more widely for desktop and server applications, which makes it easier to find components, tools, information, and support. The combination of the Linux OS with a stable TCP/IP stack has become the primary choice for many embedded systems used in many diverse applications.

3.10.1 Specific Requirements for Embedded OSs

The later chapters of this book illustrate the Linux implementation of TCP/IP in detail. An important focus of this book is embedded systems; therefore, we should take the time to discuss the requirements for a generic embedded system that seeks to host a TCP/IP stack. We can compare these requirements with specific Linux capabilities to see how Linux compares with its competitors as a choice for an embedded OS, and how well it is suited to support an embedded system's networking needs. The TCP/IP stack, like all networking protocols, is asynchronous. The stack implementation is event oriented where processing is invoked when a packet becomes available at a given layer. Each packet travels through the stack as a separate thread. In addition, there are concurrent activities needed to keep track of time- related events specified by the protocols. In the light of these general requirements, the following is a brief list of specific facilities that every embedded OS should have.

Timer facility: All protocols require timers for retransmissions and timeouts. Some of the simple OSs used for embedded systems either lack this simple facility or do not have a consistent interface. For example, Chapter 8 will discuss the use of timers in the TCP protocol.

Concurrency and multitasking: The socket API should allow for multiple simultaneous users. The TCP/IP stack should be multithreaded. At the simplest level, buffer contention should be avoided by having semaphore protection at the buffer level. Chapter 6, "The Linux TCP/IP Stack," discusses Linux kernel threading and how these threads are used within TCP/IP.

Buffer management: TCP/IP and most communication protocols are most efficient when they are provided with a fixed-length buffer system. In earlier TCP/IP implementations and those commonly used in embedded systems, the buffer pool was pre-allocated at boot-up time. This system avoids problems with heap fragmentation but is limited because the maximum number of buffers is fixed. Buffers are assigned dynamically from the pool when a new incoming packet arrives at the network interface device or a new outgoing packet is created at the socket layer. As we will see in Chapter 7, "Linux Socket Buffers and Linux Memory Allocation," Linux networking buffers are called socket buffers. Socket buffers are allocated from a slab cache, which solves the fragmentation problem. Another advantage of the slab cache is scalability and that there is no preconfigured upper bound.

Link layer facility: Complex protocol implementation requires a way to layer protocols below IP and above the network interface driver. It is desirable to have a generalized facility to add protocols such as Network Address Translation (NAT), Point-to-Point Protocol (PPP), or other software-implemented bridging or switching capability without having to alter either the IP sources or network interface driver source code. As shown in Chapter 4, Linux provides a network interface device. These devices can be "real" devices with interrupt capability or pseudo-devices that can interface TCP/IP with an underlying protocol stack. In addition, as discussed in later chapters, Linux provides a queuing layer between the network interface drivers to the network layer to handle specific requirements for multiple traffic classes.

There are also a few other facilities where Linux can be differentiated from other real-time OSs used for embedded applications. Standard Linux like other earlier Unix implementations is a nonpreemptible implementation of the OS kernel. However, since TCP/IP is part of the Linux kernel, it can make use of the softirq kernel threading facility. We will see how this works in Chapter 6, "The Linux TCP/IP Stack." Linux has the capability of deferring interrupt-level work to kernel threads to decrease latency problems.

Low latency: The operating system should not add any more latency than necessary to the minimum amount of processing required at interrupt time for the physical reception and transmission of a frame. The goal of the OS should be to minimize context switches while a packet is processed by the stack. Linux uses softirqs to handle most of the internal processing. In addition, as we will see in later chapters, fast paths are provided for packet transmission and reception to reduce the amount of overhead.

Minimal data copying: It is advantageous for embedded systems to minimize the amount of copying necessary to move a packet of data from the application, down through the stack to the transmission media. Embedded applications are performance sensitive and should have a TCP/IP stack implementation that has no buffer copying at the device driver level and allows for the option of eliminating buffer copying at the socket level. Typically, the Ethernet chip or hardware networking PHY device places the packet directly in memory via DMA. Linux provides scatter-gather DMA support where the socket buffers are set up to allow for the direct transmission of lists of TCP segments. In addition, at the user level, when data is transferred through a socket, copying can be avoided and data can be mapped directly into the user space from the kernel space.

3.11 TCP/IP Standards, Numbers, and Practical Considerations

As discussed in Chapter 1, the bodies responsible for the standards that govern the Internet are the Internet Engineering Task Force (IETF), *www.ietf.org* and the Internet Architecture Board (IAB). These bodies are part of the Internet Society (ISOC), *www.isoc.org.* The Internet standards are published as Request for Comments (RFCs). Many Web sites provide databases of RFCs; one of the best at the time of writing is *http://www.faqs.org/rfcs/.*

Throughout this book are tables of values for stuffing in various function arguments and structure fields. There are tables in each chapter for some of these values, and many of the values are actually officially assigned numbers. In some places, the lists of numbers might not be entirely complete because of new protocols that are registered from time to time. Much of what makes the Internet function is based on these agreed-upon numbers. The numbers include the protocol fields in the link layer or Ethernet header, the protocol numbers in the IP header, well-known transport layer port numbers, and many others. Figures 3.3 and 3.4 illustrate the link layer headers, and Table 3.1 shows the protocol fields for the IPv4 Ethernet MAC header. These number assignments at one time were specified by RFCs that were re-issued from time to time as new protocols were defined. The last of these RFCs that defined protocol numbers was RFC 1700. Now, as of RFC 3232 in early 2002, the responsibility of maintaining assigned numbers was removed from the RFC editor. The assigned numbers are now maintained in a database by the Internet Assigned Number Authority (IANA) *www.iana.org/assignments*.

3.12 Summary

This first part of this chapter introduced the origins of TCP/IP and the history of its implementation in Linux. Later, the TCP/IP protocol stack was laid out in terms of the OSI seven-layer model. We explored each major component of TCP/IP and how it fits into the conceptualized OSI networking model. In this chapter, some issues specific to embedded systems were discussed relative to the use and implementation of TCP/IP. We showed how Linux is an excellent platform for networking either in embedded systems or any other networking-oriented computing application. Now that you have a basic understanding of data communication protocols and the TCP/IP stack, let’s delve into some of the infrastructure in the Linux implementation of TCP/IP.

Chapter 4: Linux Networking Interfaces and Device Drivers

In Chapter 1, "Introduction," we discussed networking in general and how to build on basic networking concepts. In Chapter 2, "Broadband Networking Protocols of Yesterday and Today," we discussed networking protocols that illustrated some concepts discussed in Chapter 1 and also discussed some networking protocols that are often interfaced to TCP/IP. Chapter 3, "TCP/IP in Embedded Systems," covered the TCP/IP protocol and its implementation in general terms without specific reference to its implementation in Linux. There are some very good books on Linux device drivers, and we won't attempt to replicate that work here. However, we will discuss in detail the mechanisms specific to Linux network interface drivers. This is because a good understanding of the Linux TCP/IP stack requires a good understanding of the architecture of the network interface devices, how they function in the operating system (OS), and how they interface to IP and other network layer protocols.

4.1 Introduction

In this chapter, we show the internal structure of Linux network interface drivers. Later chapters continue this discussion by showing the infrastructure of Linux above the network interface drivers and how the drivers are interfaced to IP and other network layer protocols. Much of the Linux internal kernel implementation is not completely documented. There are several good reference books for writers of Linux device drivers, such as [Rubini00], which is excellent but does not discuss the networking interface drivers as much as other drivers and the details of how the drivers interface to the TCP/IP stack. This chapter does not attempt to duplicate other works and therefore will not provide a complete discussion about writing Linux device drivers. Instead, the intent of this book is to discuss the networking interface drivers and in particular, their internal data structures, and how data packets flow into and out of the TCP/IP stack. It is quite possible to construct a device driver by using an existing device driver as sample code. This is often the quickest way to approach a development task. However, it is helpful for driver writers, and kernel diggers in general, to have a good understanding of how the network interface driver is structured.

This chapter also covers the network driver initialization and registration process, and the programming interface required by network drivers for the driver to function properly in the OS. In other Unix-like operating systems, the interface functions are often called the *driver entry points*. However, in Linux, these driver interface functions are called *network interface service functions*. We will also discuss the packet queuing layer and how the packets are transmitted and received, although some aspects of how packets are routed through the queuing layer are in other chapters. For example, other important aspects of the packet queuing layer such as the capability to work with multiple queuing disciplines and traffic class-based schedulers are discussed in later chapters along with IP routing.

In the most basic sense, the Linux networking interface drivers are similar to network drivers used with other Unix-like operating systems. The device driver details are hidden from the network layer implementation. The TCP/IP stack provides a registration mechanism between the

device drivers and the layer above, and this registration mechanism allows the output routines in the networking layer, such as IP, to call the driver's transmit function for a specific interface port without needing to know the driver's internal details. Another similarity between the Linux approach and other operating systems is that network interface drivers only interface to other parts of the Linux kernel. There is no direct Application Programming Interface (API) with which code outside the kernel can make direct calls to the network driver. In addition, unlike other drivers, network drivers are not accessed via standard IO through file descriptors. Instead, packets are transmitted and received by Layer 3 protocols which are connected to the driver via a registration mechanism. Moreover, network interface driver configuration is not done via direct IO; instead, driver configuration is done either with ioctl calls through the socket API or via the sysctl interface.

4.2 Network Interface Devices

Each network interface device must have a driver, and the driver provides initialization functions that are called at kernel startup or module load time. The driver also includes tables that allow the device to work with the Linux kernel. All Linux network interface devices can be associated with the kernel in two different ways, either at kernel build time or later by loading once the kernel is up and operating. In addition, a Linux network interface driver can be implemented as a module so it is not necessary to statically link it into the kernel, and generally, it is preferable to implement the driver as a module. If the driver is a module, it does not preclude it from being statically linked with the kernel. Whether modules are statically linked into the kernel is specified at kernel configuration time Driver modules can be loaded dynamically at boot time or later using the insmod(8) command. Whether the driver is written as a module or is statically linked, the driver must provide an initialization function that is called by the kernel before the network interface device is ready to receive and transmit packets. Although primarily about kernel versions before 2.6, [Rubini00a] has a good discussion of how to implement kernel modules. The network driver initialization function sets up the driver's internal data structures, the driver is introduced to the Linux kernel, and the send and receive queues are connected to the TCP/IP stack. The main driver structure is the net_device structure. Although there are many different types of network interface drivers, they each have one thing in common: they must initialize an instance of the net device structure. The initialization sequence for a driver for a PCI bus device is explained later in the text.

4.3 The Network Device Structure, net_device

The net device structure, defined in file *linux/include/netdevice.h*, is the data structure that defines an instance of a network interface. It tracks the state information of all the network interface devices attached to the TCP/IP stack. The structure might seem longer and more complex than is necessary, but it contains fields for implementing devices that have been added after the more generic network interfaces were originally defined. In our description of the net_device structure, the fields that are of most interest to network interface driver writers contain explanations. We also include explanations of the things needed to connect the network interface driver to the TCP/IP stack. In general, we concern ourselves only with the public part of the driver's data structure, but some key fields in the private part of the driver's data structure are important for PCI devices. This is because they must be initialized with a pointer to the PCI card's configuration space, and this pointer is de-referenced later when the driver allocates buffer

space for Ethernet DMA that is mapped into PCI device space. However, most of the discussion in this section pertains to the public part of the structure contained within the net device structure.

```
struct net_device
{
```

This first field, name, is the beginning of the visible part of this structure. It contains the string that is the name of the interface. By *visible*, we mean that this part of the data structure is generic and doesn't contain any private areas specific to a particular type of device. This is the part that is initialized by the Space.c file, which contains the generic probe functions for each device type.

```
    char                name[IFNAMSIZ];
```

Mem_end is the pointer to the end of the device-specific shared memory area, and mem_start is the pointer to the beginning of the area.

```
    unsigned long       mem_end;
    unsigned long       mem_start;
```

Base_addr is the device I/O address used primarily by x86 processors and other architectures that do memory mapped I/O. IRQ is the device's Interrupt number, also used by x86 processors.

```
    unsigned long       base_addr;
    unsigned int        irq;
```

Both of the following fields are used by some types of hardware but are generally specified by the file, *linux/drivers/net/Space.c*, which holds the initial configuration information for various classes of devices.

```
    unsigned char       if_port;
    unsigned char       dma;
```

This field, state holds the network device state. These values are private to the queuing layer and are not used anywhere else. They are shown in Table 4.1.

Table 4.1: Network Queuing Layer Device State

State	Value	Description
__LINK_STATE_XOFF	0	Device queue turned off.
__LINK_STATE_START	1	Device queue turned on.
__LINK_STATE_PRESENT	2	Device ready to be scheduled.
__LINK_STATE_SCHED	3	
__LINK_STATE_NOCARRIER	4	
__LINK_STATE_RX_SCHED	5	Device placed on poll list.
__LINK_STATE_LINKWATCH_PENDING	6	

```
    unsigned long       state;
```

```
struct net_device   *next;
```

Init is the pointer to the device's initialization function. This function is called only once. This is the last field that is pre-initialized by the probe functions in *linux/drivers/net/Space.c*.

```
int                 (*init)(struct net_device *dev);

struct net_device    *next_sched;
```

Ifindex is the interface index, and iflink is the unique device identifier.

```
int                 ifindex;
int                 iflink;
```

The next field, net_device_stats, is a pointer to the driver's function for returning the network interface statistics including the various counters. Iw_statistics is for wireless devices and returns a set of interface statistics specific for wireless interfaces.

```
struct net_device_stats* (*get_stats)(struct net_device *dev);
struct iw_statistics*    (*get_wireless_stats)(struct net_device *dev);
```

The next field points to a list of functions to handle device extensions that are specific to wireless devices. The file *iw_handler.h* has more details of these functions, which are an alternative to using ioctl.

```
struct iw_handler_def *  wireless_handlers;
struct ethtool_ops       *ethtool_ops;
```

Now we have come to the end of the visible part of the net_device structure. All the fields that follow are internal and might change in future revisions of the kernel; therefore, device driver implementers should not count on the following fields.

The next few fields can be used for auto-powerdown code. Trans_start is the timestamp in jiffies of the last transmitted packet, and last_rx is the timestamp of the last received packet. Jiffies contains the current time in ticks.

```
unsigned long        trans_start;
unsigned long        last_rx;
```

The next field, flags, contains the BSD style interface flags.

```
unsigned short       flags;    /* interface flags (a la BSD)*/
unsigned short       gflags;
```

This field, priv_flags, is like flags but is private and will never show up at the user level.

```
unsigned short       priv_flags;
unsigned short       unused_alignment_fixer;
```

Mtu contains the interface's Maximum Transmission Unit (MTU) value. The next field, type, is the interface hardware type.

```
unsigned            mtu;
unsigned short      type;
```

Hard_header_len is the size of the link layer header (sometimes called the hardware header in the Linux sources) for this network interface. Priv is the pointer to the driver's private area. We will see how this is initialized in the next section, and master is the pointer to the master device for this device's group.

```
unsigned short      hard_header_len;
mvoid               *priv;
struct net_device   *master;
```

The next few fields contain the interface specific link layer address information. Broadcast is the link layer broadcast address, dev_addr is the interface's specific hardware address, and addr_len is the length of the hardware address.

```
unsigned char       broadcast[MAX_ADDR_LEN];
unsigned char       dev_addr[MAX_ADDR_LEN];
unsigned char       addr_len;
```

Mc_list is the list of the network interface's link layer or MAC multicast addresses, and mc_count is the number of multicast addresses on the list. If set to TRUE, the field promiscuity indicates that promiscuous mode is set for this interface. Promiscuous mode was discussed in earlier chapters. It allows an interface to see all packets on the wire.

```
struct dev_mc_list  *mc_list;
int                 mc_count;
int                 promiscuity;
int                 allmulti;
```

The following two fields are for a watchdog timer, if one is used by this interface.

```
int                 watchdog_timeo;
struct timer_list   watchdog_timer;
```

The next few fields are protocol-specific pointers. Atalk_ptr is a pointer to AppleTalk-specific data, and ip_ptr is a pointer to IPv4-specific data. Dn_ptr is for DECnet-specific data, and ip6_ptr is for IPv6 data. Finally, ec_ptr is Econet specific, and ax25_ptr is specific to the AX.25 protocol.

```
void                *atalk_ptr;
void                *ip_ptr;
void                *dn_ptr;
void                *ip6_ptr;
void                *ec_ptr;
void                *ax25_ptr;
```

The next field, poll_list, is a pointer to the list of queued packets. This field and the next group of fields are used by the queuing layer to allow for multiple queuing disciplines to be used with the network interface device. Quota is used by the backlog device (blog_dev) for input queuing, and weight is the backlog weight also used by blog_dev for input packets.

```
    struct list_head    poll_list;
    int                 quota;
    int                 weight;
```

Qdisc is the queue discipline that is used for this network interface.

```
    struct Qdisc        *qdisc;
    struct Qdisc        *qdisc_sleeping;
```

Qdisc_sleeping points to the list of registered queue disciplines used with this network interface device, and the next field, qdisc_ingress, is the queuing dis cipline for input packets.
Tx_queue_len is the maximum number of output packets allowed for this device queue.

```
    struct Qdisc        *qdisc_list;
    struct Qdisc        *qdisc_ingress;
    unsigned long       tx_queue_len;
```

The next two fields are the spinlock for synchronizing the driver's hard_start_xmit function when used with multiple CPU applications. Queue_lock is the device queue mutex lock.

```
    spinlock_t          xmit_lock;
    int                 xmit_lock_owner;
    spinlock_t          queue_lock;
```

Refcnt contains the number of references to this device.

```
    atomic_t            refcnt;
```

Todo_list is for delayed device registration and unregistration, and reg_state is the state machine for delayed registration and unregistration.

```
    struct list_head    todo_list;
    enum {  NETREG_UNINITIALIZED=0,
```

Register_netdevice has been called.

```
        NETREG_REGISTERING,
```

Registration is completed.

```
        NETREG_REGISTERED,
```

Unregistration has begun.

```
        NETREG_UNREGISTERING,
```

Unregistration is completed.

```
        NETREG_UNREGISTERED,
```

The function free_netdev is called to free the net_device structure.

```
        NETREG_RELEASED,
    }  reg_state;
```

The following field contains the network interface device features. The features are listed in Table 4.2.

```
    int                 features;
```

Table 4.2: Bit Definitions for the Features Field in the Net_device Structure

Name	Value	Purpose
NETIF_F_SG	0x1	Set if device supports scatter/gather IO, which means that the device can transmit multiple buffers with a single call to hard_start_xmit().
NETIF_F_IP_CSUM	0x2	This bit is set if the device hardware can only checksum TCP/UDP over IPv4.
NETIF_F_NO_CSUM	0x4	Set if the device does not require checksum such as the loopback device.
NETIF_F_HW_CSUM	0x8	Set if the device can checksum all the packets.
NETIF_F_HIGHDMA	0x20	Set if the device can DMA to high memory.
NETIF_F_FRAGLIST	0x40	Another flag for scatter/gather IO. It indicates that IP fragmentation is possible on this device.
NETIF_F_HW_VLAN_TX	0x80	This device supports transmit VLAN hardware acceleration.
NETIF_F_HW_VLAN_RX	0x100	The device supports receive VLAN hardware acceleration.
NETIF_F_HW_VLAN_FILTER	0x200	The device supports receive filtering on VLAN.
NETIF_F_VLAN_CHALLENGED	0x400	Indicates that the device cannot handle VLAN packets.
NETIF_F_TSO	0x800	The device can offload TCP/IP segmentation.

This function, uninit, is called after the device is detached from network, and destructor is called after the last user reference is removed.

```
    void                (*uninit)(struct net_device *dev);
    void                (*destructor)(struct net_device *dev);
```

Next in the net_device structure, we have the pointers to the service routines for the network interface device. These functions are discussed in detail in Section 4.7.

```
    int                 (*open)(struct net_device *dev);
    int                 (*stop)(struct net_device *dev);
    int                 (*hard_start_xmit) (struct sk_buff *skb,
                         struct net_device *dev);
#define HAVE_NETDEV_POLL
    int                 (*poll) (struct net_device *dev, int *quota);
    int(*hard_header)   (struct sk_buff *skb,
                         struct net_device *dev,
                         unsigned short type,
                         void *daddr,
                         void *saddr,
                         unsigned len);
    int                 (*rebuild_header)(struct sk_buff *skb);
#define HAVE_MULTICAST
    void                (*set_multicast_list)(struct net_device *dev);
#define HAVE_SET_MAC_ADDR
    int                 (*set_mac_address)(struct net_device *dev,
                         void *addr);
#define HAVE_PRIVATE_IOCTL
    int                 (*do_ioctl)(struct net_device *dev,
                         struct ifreq *ifr, int cmd);
#define HAVE_SET_CONFIG
    int                 (*set_config)(struct net_device *dev,
                         struct ifmap *map);
#define HAVE_HEADER_CACHE
    int                 (*hard_header_cache)(struct neighbour *neigh,
                         struct hh_cache *hh);
    void                (*header_cache_update)(struct hh_cache *hh,
                         struct net_device *dev,
                         unsigned char *  haddr);
#define HAVE_CHANGE_MTU
    int                 (*change_mtu)(struct net_device *dev, int new_mtu);

#define HAVE_TX_TIMEOUT
    void               (*tx_timeout) (struct net_device *dev);

    void               (*vlan_rx_register)(struct net_device *dev,
                        struct vlan_group *grp);
    void               (*vlan_rx_add_vid)(struct net_device *dev,
                        unsigned short vid);
    void               (*vlan_rx_kill_vid)(struct net_device *dev,
                        unsigned short vid);

    int                (*hard_header_parse)(struct sk_buff *skb,
                        unsigned char *haddr);
    int                (*neigh_setup)(struct net_device *dev, struct
                        neigh_parms *);
    int                (*accept_fastpath)(struct net_device *, struct
                        dst_entry*);
```

The next field, br_port, is for implementing Layer 2 bridging.

```
    struct net_bridge_port    *br_port;
```

The next two fields are used only if fast routing is configured.

```
#ifdef CONFIG_NET_FASTROUTE
```

```
#define NETDEV_FASTROUTE_HMASK 0xF
```

This is a semiprivate area and should remain near the bottom of the net_device structure.

```
    rwlock_t            fastpath_lock;
    struct dst_entry    *fastpath[NETDEV_FASTROUTE_HMASK+1];
#endif
#ifdef CONFIG_NET_DIVERT
```

This field, divert, will be initialized by each interface type initialize routine.

```
    struct divert_blk   *divert;
#endif /* CONFIG_NET_DIVERT */
```

The next field, class_dev, contains device class information, including the class, net, and name. See *linux/include/linux/device.h* for more information.

```
    struct class_device class_dev;
    struct net_device_stats* (*last_stats)(struct net_device *);
} ;
```

4.4 Network Device Initialization

Network device initialization begins when the Linux kernel “discovers” a device of a certain type by calling a probe function looking for a match of a particular hardware interface with its associated driver.

Once a match is discovered, the driver’s specific initialization function is called. This function is generally typed __devinit or __init, both of which are defined in file *linux/include/linux/init.h*. When the network interface driver’s initialization or probe function is called, the first thing it does is allocate the driver’s private data structure, which also contains the generic net_device structure. Next, it must set a few key fields in the structure, the first of which is the name field, which is used later to register the network interface device. Name is generally set to a string of characters that identifies the class of device. For example, in Ethernet devices, the name field begins with the string "eth" . The last characters in the name field should be set to the format string %d. Later, when the device is registered, the formatting string is replaced with a number from zero to 99. These first two initialization steps can be done in at least three ways. Device drivers for non-Ethernet devices can set up the data structure directly in the driver’s probe function by calling kmalloc to allocate the structure, calling dev_alloc_name to set up the name string, and then directly initializing the other device-specific fields in the net_device structure. See Figure 4.1 for an illustration of three initialization methods of the net device structure used by various network interface drivers.

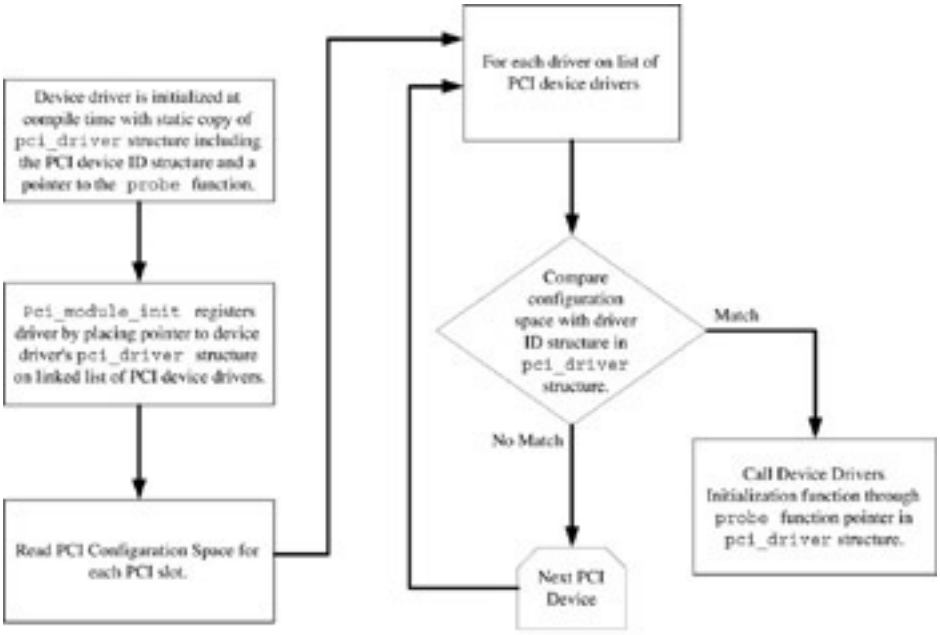
Device driver is initialized at compile time with static copy of pci_driver structure including the PCI device ID structure and a pointer to the probe function.
pci_module_init registers driver by placing pointer to device driver's pci_driver structure on linked list of PCI device drivers.
Read PCI Configuration Space for each PCI slot.
For each driver on list of PCI device drivers
Compare configuration space with driver ID structure in pci_driver structure.
Match
No Match
Call Device Drivers Initialization function through probe function pointer in pci_driver structure.
Next PCI Device

Figure 4.1: Network interface driver registration sequence.

Linux provides some generic device allocation and initialization functions for the convenience of writers of network interface drivers. Most of these are found in *linux/net/net_init.c*, except for some functions that are specific for Ethernet devices. For example, we will look at the initialization sequence for an Ethernet network interface device, shown in Figure 4.1. This sequence uses an allocation method provided for Ethernet drivers. The function alloc_etherdev defined in *drivers/net/net_init.c* and declared in file *linux/include/linux/etherdevice.h*, does both structure allocation and name string initialization. Under the covers, alloc_etherdev calls alloc_netdev and passes it a pointer to a setup function as the second argument. This function used to be ether_setup in earlier kernel versions. At this point, the function netdev_boot_setup, actually saves any boot time settings in an instance of the netdev_boot_setup structure. Later, these settings are put into the net_device structure by calling netdev_boot_setup_check.

At this point in the initialization sequence, the net_device structure is initialized with the name, the irq, base address, and bounds of the IO mapped memory. However, for most devices more initialization must be done before the network interface can be registered. Interrupt hardware must be set up, and any device- specific information is read from the device's registers and PCI configuration space if applicable. After this is complete, the initialization function sets up spin locks, and some additional hardware-specific information. If these steps don't generate an error, the field priv is set to point back to the private part of the data structure, and buffers are allocated for transmit and receive space from memory that is accessible by DMA. Next, pointers to the driver's service routines are set in the net_device structure. The most important service routines to support are the open, hard_start, stop, get_stats, set_multicast_list, and do_ioctl functions. The network interface service routines are discussed in more detail in Section 4.3. Next, the features field of the net device structure is filled in to indicate device abilities. Drivers for network interface devices that have hardware speed-up capabilities, such as the capability to calculate IP checksums, will indicate it in this field.

4.4.1 Initialization of the Net Device Structure

Each network driver provides an initialization function, which is called after any module housekeeping and PCI device ID matching is complete. For example, PCI device initialization is discussed in detail later. As we shall see, in the case of PCI network interface devices, the driver's initialization function is called by de-referencing the probe field in the pci_driver structure. The initialization function in each driver must allocate the net_device structure, which is used to connect the network interface driver with the network layer protocols. Typically, most drivers have unique hardware-specific data to maintain, so the driver will allocate a larger private structure, which in turn contains the net_device structure. For example, Ethernet drivers can call the function alloc_etherdev, declared in the file *ether*[*device.h*] and defined in [*net_init.c*]. This function is provided by the Linux kernel as a support routine for network device structure allocation and initialization. Alternatively, non-Ethernet device drivers may call kmalloc directly to allocate the data structure and then initialize it by "hand." For an example of Ethernet interface initialization, look at the following code example where we show the initialization function from the e100 driver, called e100_found1. It allocates the public dev structure and the driver's private data structure at the same time by calling alloc_etherdev. It sets its private structure bbp by accessing it through the priv field in the net_device. Next, it sets the network service function entry points in the dev structure.

```
static int __devinit e100_found1(struct pci_dev *pcid,
const struct pci_device_id *ent)
{
    static int first_time = true;
    struct net_device *dev = NULL;
    struct e100_private *bdp = NULL;
    int rc = 0;
    u16 cal_checksum, read_checksum;
```

The structure e100_private is declared in file *linux/drivers/net/e100/*[*e100.h*]. Actually, alloc_etherdev is one of several allocation functions provided for different classes of network devices. The argument to alloc_etherdev is the size of the private structure, but alloc_etherdev also allocates enough space for the net_device structure, too. It sets the priv field in the net_device structure to point to the private structure.

```
    dev = alloc_etherdev(sizeof (struct e100_private));
    if (dev == NULL) {
        printk(KERN_ERR "e100: Not able to alloc etherdev struct\ n");
        rc = -ENODEV;
        goto out;
    }
    SET_MODULE_OWNER(dev);
    if (first_time) {
        first_time = false;
        printk(KERN_NOTICE "%s - version %s\ n",
               e100_full_driver_name, e100_driver_version);
        printk(KERN_NOTICE "%s\ n", e100_copyright);
        printk(KERN_NOTICE "\ n");
    }
```

Bdp points to the device's private data structure and, as shown in the preceding code, space for the structure was already allocated when we called alloc_etherdev.

```
bdp = dev->priv;
bdp->pdev = pcid;
bdp->device = dev;
pci_set_drvdata(pcid, dev);
SET_NETDEV_DEV(dev, &pcid->dev);
```

Here, we set some fields and initialize timers specific to this driver's type of Ethernet device.

```
bdp->flags = 0;
bdp->ifs_state = 0;
bdp->ifs_value = 0;
bdp->scb = 0;
init_timer(&bdp->nontx_timer_id);
bdp->nontx_timer_id.data = (unsigned long) bdp;
bdp->nontx_timer_id.function = (void *) &e100_non_tx_background;
INIT_LIST_HEAD(&(bdp->non_tx_cmd_list));
bdp->non_tx_command_state = E100_NON_TX_IDLE;
init_timer(&bdp->watchdog_timer);
bdp->watchdog_timer.data = (unsigned long) dev;
bdp->watchdog_timer.function = (void *) &e100_watchdog;
```

This device is on the PCI bus.

```
if ((rc = e100_pci_setup(pcid, bdp)) != 0) {
    goto err_dev;
}
```

The function e100_alloc_space mostly allocates the memory for DMA, but it also initializes some fields in the private data structure.

```
if ((rc = e100_alloc_space(bdp)) != 0) {
    goto err_pci;
}
```

Here, we check for matching device IDs and revisions.

```
if (((bdp->pdev->device > 0x1030)
     && (bdp->pdev->device < 0x103F))
     || ((bdp->pdev->device >= 0x1050)
     && (bdp->pdev->device <= 0x1057))
     || (bdp->pdev->device == 0x2449)
     || (bdp->pdev->device == 0x2459)
     || (bdp->pdev->device == 0x245D)) {
    bdp->rev_id = D101MA_REV_ID;/* workaround for ICH3 */
    bdp->flags |= IS_ICH;
}
if (bdp->rev_id == 0xff)
    bdp->rev_id = 1;
if ((u8) bdp->rev_id >= D101A4_REV_ID)
    bdp->flags |= IS_BACHELOR;
if ((u8) bdp->rev_id >= D102_REV_ID) {
```

```
        bdp->flags |= USE_IPCB;
        bdp->rfd_size = 32;
    }  else {
        bdp->rfd_size = 16;
    }
    dev->vlan_rx_register = e100_vlan_rx_register;
    dev->vlan_rx_add_vid = e100_vlan_rx_add_vid;
    dev->vlan_rx_kill_vid = e100_vlan_rx_kill_vid
    dev->irq = pcid->irq;
    dev->open = &e100_open;
    dev->hard_start_xmit = &e100_xmit_frame;
    dev->stop = &e100_close;
    dev->change_mtu = &e100_change_mtu;
    dev->get_stats = &e100_get_stats;
    dev->set_multicast_list = &e100_set_multi;
    dev->set_mac_address = &e100_set_mac;
    dev->do_ioctl = &e100_ioctl;
    if (bdp->flags & USE_IPCB) {
        dev->features = NETIF_F_SG | NETIF_F_IP_CSUM |
                        NETIF_F_HW_VLAN_TX | NETIF_F_HW_VLAN_RX;
```

Once the net_device structure is initialized, we can register it. This is where the network interface driver is hooked to the TCP/IP stack. This registration function and other utility functions for network interface registration are discussed later in this chapter.

```
    if ((rc = register_netdev(dev)) != 0) {
        goto err_dealloc;
    }
```

Now that we are registered, the rest of the work done by this initialization function is specific to this driver's Ethernet chip.

```
    e100_check_options(e100nics, bdp);
    if (!e100_init(bdp)) {
        printk(KERN_ERR "e100: Failed to initialize, instance #%d\ n",
e100nics);
        rc = -ENODEV;
        goto err_unregister_netdev;
    }
```

Check to see if checksum is valid.

```
    cal_checksum = e100_eeprom_calculate_chksum(bdp);
    read_checksum = e100_eeprom_read(bdp, (bdp->eeprom_size - 1));
    if (cal_checksum != read_checksum) {
        printk(KERN_ERR "e100: Corrupted EEPROM on instance #%d\ n",
               e100nics);
        rc = -ENODEV;
        goto err_unregister_netdev;
    }
    e100nics++;
    e100_get_speed_duplex_caps(bdp);
    printk(KERN_NOTICE
           "e100: %s: %s\ n",
           bdp->device->name, "Intel(R) PRO/100 Network Connection");
```

```
    e100_print_brd_conf(bdp);
    bdp->wolsupported = 0;
    bdp->wolopts = 0;
    if (bdp->rev_id >= D101A4_REV_ID)
        bdp->wolsupported = WAKE_PHY | WAKE_MAGIC;
    if (bdp->rev_id >= D101MA_REV_ID)
        bdp->wolsupported |= WAKE_UCAST | WAKE_ARP;
```

Check if WoL is enabled on EEPROM.

```
    if (e100_eeprom_read(bdp, EEPROM_ID_WORD) & BIT_5) {
```

Magic Packet WoL is enabled on device by default if EEPROM WoL bit is TRUE.

```
        bdp->wolopts = WAKE_MAGIC;
    }
    printk(KERN_NOTICE "\ n");
    goto out
    err_unregister_netdev:
```

If we got an error during hardware initialization, we must un-register the device.

```
    unregister_netdev(dev);
err_dealloc:
    e100_dealloc_space(bdp);
```

We end up here if we received an error while attempting the PCI specific initialization.

```
err_pci:
    iounmap(bdp->scb);
    pci_release_regions(pcid);
    pci_disable_device(pcid);
err_dev:
    pci_set_drvdata(pcid, NULL);
    kfree(dev);
out:
    return rc;
}
```

4.5 Device Discovery and Dynamic Network Interface Driver Initialization

We don’t attempt to explain every type of hardware supported by the Linux operating system. However, it is useful to see how a network interface driver is automatically configured and how a device is matched by PCI ID to the correct driver. We use a PCI device as an example because most common network interface devices including Ethernet chips are implemented as PCI devices. This section explains what happens for one method of discovering a PCI device. We examine how a PCI device is matched and the driver’s initialization function is called. In general, a device is detected on the PCI bus by matching the vendor and device IDs in the PCI configuration space, on the PCI card with an internally stored value in the network device driver.

The main match is on the PCI device ID and vendor codes, both of which are 16-bit numbers, but the PCI vendor and class can be matched, too.

For an example, we will use the IDs for one of the Intel Ethernet chips handled by the e100 driver. These values are shown in Table 4.3. We will describe the things that happen when a device is matched. Figure 4.2 shows a typical initialization sequence for a network interface driver module for a PCI hardware device.

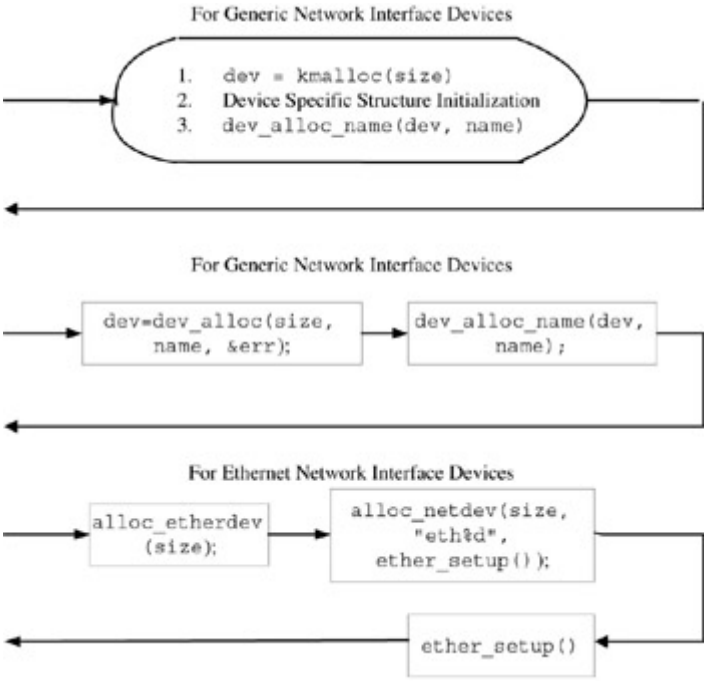

Figure 4.2: PCI device driver initialization sequence.

Table 4.3: PCI Device ID Structure for an Ethernet Chip

Field	Purpose	Example	Example Description
vendor	Vendor ID	0x8086	Intel
device	Device ID	0x1229	Intel 82556 Ethernet chip
subvendor	Optional Vendor ID	0xffff	PCI_ANY_ID
subdevice	Optional Device ID	0xffff	PCI_ANY_ID
class	Class ID	0x0	
class_mask	Class mask	0x0	

Let’s begin with the module initialization before the hardware is discovered. Each network driver, implemented as a module, must contain a driver-specific module initialization function, which is called when the kernel loads the network driver module either at boot time or later. For example, we will be using the e100 Ethernet driver. We will look at the module initialization function e100_init_module in file *linux/drivers/net/e100/e100_main.c*. To begin the initialization process, if the e100 driver is statically linked into the kernel, e100_init_module is called by the kernel. However, if the driver is a module, this function is called when the insmod(8) or modprob(8) utilities are run to install the module. In our example, the driver name is initialized to "e100",

id_table is initialized to e100_id_table, and probe to the function e100_found1. E100_found1 is the function that actually does the PCI device match.

```
static int __init e100_init_module(void)
{
    int ret;
```

The first thing we do is call the generic function, pci_module_init, which does most of the actual work of registering the PCI device.

```
    ret = pci_module_init(&e100_driver);
    if(ret >= 0) {
#ifdef CONFIG_PM
        register_reboot_notifier(&e100_notifier_reboot);
#endif
    }
    return ret;
}
```

Pci_module_init is defined in *linux/include/linux/pci.h*. It places a pointer to this particular PCI driver on a linked list of PCI drivers, and this is how the device driver registers both the PCI device ID and the vendor code with the Linux kernel. Later, the ID and vendor are matched when a PCI device is discovered or plugged in (if it is hot-pluggable).

```
inline int pci_module_init(struct pci_driver *drv);
```

The only thing this function does is call pci_register_driver, defined in the file *linux/drivers/pci.pci_driver.c* to register the driver, initialize the common fields in the pci_driver structure, and add the driver to the list of registered drivers. Pci_register_driver returns the number of PCI devices that are claimed by the driver during registration, and even if its function returns a zero, the driver remains registered.

```
pci_register_driver(struct pci_driver *drv)
{
    int count = 0;
```

We initialize the common fields in the pci_driver structure including the probe field, which is called when a matching device is discovered on the PCI bus. Notice that the field probe does not point directly to e100_found1. Instead, it points to a generic pci probe function, pci_device_probe, which in turn calls e100_found1 through the probe field of drv. when the device is matched.

```
    drv->driver.name = drv->name;
    drv->driver.bus = &pci_bus_type;
    drv->driver.probe = pci_device_probe;
    drv->driver.remove = pci_device_remove;
    drv->driver.kobj.ktype = &pci_driver_kobj_type;
    pci_init_dynids(&drv->dynids);
```

This is where we actually register the driver with the kernel.

```
    count = driver_register(&drv->driver);

    if (count >= 0) {
        pci_populate_driver_dir(drv);
    }

    return count ? count : 1;
}
```

One of the fields in the pci_driver structure, probe, points to a function containing the driver's initialization entry point, which is called after a match is detected between the PCI device's configuration space and the driver's device ID structure. One of the PCI ID matches for the e100 driver is shown in Table 4.3. The sequence of events for this initialization is shown in Figure 4.2. Another way to show how this sequence works is to look at the code from one of the Ethernet drivers in Linux. For our example, we will look at the e100 driver. Once this sequence of steps is complete, the driver is matched to its device and the driver initialization can proceed.

A module for a PCI driver statically defines an instance of the pci_driver structure.

```
struct pci_driver {
```

The first field, node, is the head of the list of all the PCI drivers.

```
    struct list_head    node;
```

Name is the driver name string. Id_table is the pointer to the PCI configuration space information and must not be NULL for the probe function to be called. One example of the id_table entry for one of the Ethernet chips supported by our e100 driver is shown in Figure 4.2. Probe points to the function that is called when a matching device is discovered or, if the device is hot-pluggable, when it is inserted. Remove must not be NULL for hot-plug devices.

```
    char                *name;
    const struct pci_device_id *id_table;
    int  (*probe) (struct pci_dev *dev, const struct pci_device_id *id);
```

The next function, remove, is called when a hot-swap capable device is removed from the PCI bus. Suspend is called when a device is suspended, and resume is called when a device is woken up.

```
    void (*remove) (struct pci_dev *dev);
    int  (*suspend) (struct pci_dev *dev, u32 state);
    int  (*resume) (struct pci_dev *dev);
```

The next field, enable_wake, points to the function that enables or disables the wake_up event.

```
    int  (*enable_wake) (struct pci_dev *dev, u32 state, int enable);
    struct device_driver      driver;
    struct pci_dynids         dynids;
} ;
```

The pci_driver structure is initialized as a static structure. Here we show that the e100 module initializes the fields in the pci_driver structure. This is done in the file *linux/drivers/net/e100/e100_main.c*.

```
MODULE_DEVICE_TABLE(pci, e100_id_table);
static struct pci_driver e100_driver = {
    name:               "e100",
    id_table:           e100_id_table,
```

We can see that the probe field is initialized to e100_found1, which is called when the device is discovered or plugged in (if it is hot-pluggable).

```
    probe:              e100_found1
    remove:             __devexit_p(e100_remove1)
#ifdef CONFIG_PM
    suspend:            e100_suspend,
    resume:             e100_resume,
#endif
} ;
```

4.6 Network Interface Registration

After all the initialization of the net device structure and the associated private data structure is complete, the driver can be registered as a networking device. Network device registration consists of putting the driver's net device structure on a linked list. The files *linux/net/core/dev.c* and *linux/net/net_init.c* provide utility functions to do the registration and perform other tasks. Most of the functions involved in network device registration use the name field in the net_device structure. This is why driver writers should use the dev_alloc_name function to ensure that the name field is formatted properly. The list of net devices is protected by the netlink mutex locking and unlocking functions, rtnl_lock and rtnl_unlock. The list of devices should not be manipulated without locking because if the locks are not used, it is possible for the device list to become corrupted or two devices that try to register in parallel to be assigned the same name. The device interface utility functions for manipulating the list of network devices are shown later in this section. Each function described here takes a mutex lock so it is safe to call. There are also unsafe versions of each of these functions defined in *linux/core/dev.c* (in most cases) that can be used if the caller takes the netlink semaphore. The unsafe versions generally have the same name but proceeded by a double underscore with the exception of the register function, which has an unsafe version is called register_netdevice. The network interface driver registration sequence was shown earlier in Figure 4.1.

To perform the last step of network interface registration, the driver's initialization function calls register_netdev, in the file *net_init.c*.

```
int register_netdev(struct net_device *dev)
{
    int     err;
```

We lock by taking the netlink semaphore.

```
    rtnl_lock();
```

Dev_alloc_name is called to substitute the formatting string, %d, in the device name with a number. The number selected is the total number of devices registered so far for the particular type supported by this driver minus one. For example, the first Ethernet device will be called eth0, the second one will be eth1, and so on.

```
    if (strchr(dev->name, '%'))
    {
        err = dev_alloc_name(dev, dev->name);
        if (err < 0)
            goto out;
    }
```

This is a backward compatibility hook for implicit allocation of old Ethernet devices and may be deprecated. By the time this function is called, the name field in the net_device structure should be initialized to the base name plus the formatting character.

```
    if (dev->name[0]==0 || dev->name[0]==' ')
    {
        err = dev_alloc_name(dev, "eth%d");
        if (err < 0)
            goto out;
    }
```

We call register_netdevice to do the heavy lifting associated with network interface registration. If the caller knows that the device name is already set up properly, register_netdevice can be called directly, but only if the netlink semaphore is taken first.

```
    err = register_netdevice(dev);
out:
```

We must unlock before returning.

```
    rtnl_unlock();
    return err;
}
```

Linux provides the capability of using one of a number of multiple queuing disciplines and class-based scheduling methods for drivers that can benefit from a performance increase. See Chapter 6 for a detailed discussion on the queue dis ciplines and packet scheduling mechanisms. The netlink layer is an internal message-based communication facility, which is discussed in Chapter 5, "Linux Sockets." It allows application programs to use the socket API to set and get information about each of the attached protocols. Chapter 5 includes a discussion of how to use the netlink type sockets to control and configure the protocols from the application layer. Next, the default packet scheduler and queue disciplines are set for this device by calling dev_init_scheduler in the file *sch_generic.c*. (Theoretically, although this isn't done very often, the packet scheduler and queue disciplines could be changed at any time as long as the driver is off line and the appropriate lock mutexes are taken.) The function dev_init_scheduler sets the qdisc field of the net device structure to point to the noop queue discipline. A driver can change the queue discipline later when the driver's open function is called, but in the meantime, the

qdisc and qdisc_sleeping fields should point to a default queuing discipline as the net device structure is initialized.

4.6.1 Network Device Registration Utility Functions

All of the functions in this section are declared in the file *linux/include/linux/netdevice.h*. The first function dev_get_by_name finds a device by name. It can be called from any context because it does its own locking. It returns a pointer to a net_device based on the string name. The reference count field in net_device is incremented before the function returns.

```
struct net_device *dev_get_by_name(const char *name);
```

The first function, register_netdev, registers a network interface device driver. This version of the function acquires the lock. The function register_netdevice can also be used but it requires the caller to hold the rtnl lock.

```
int         register_netdev(struct net_device *dev);
```

An important responsibility of register_netdevice is to place the net device structure on the list of network device drivers maintained in the Linux kernel. Before doing this, it initializes the device's queue_lock and xmit_lock fields. It then checks a global variable called dev_boot_phase, set at compile time to indicate that no devices are initialized. If dev_boot_phase is set, the net_dev_init function is called, which initializes the queuing layer—this will only happen once as the first network device is initialized because dev_boot_phase is reset to zero when net_dev_init exits. (For more details about the queuing layer, see Chapter 6.) A unique ifindex is assigned to the device, and this number is set in the ifindex field of the netdevice. Next, the state field is set to __LINK_STATE_PRESENT, which indicates that the device is present and ready to be scheduled.

```
int         register_netdevice(struct net_device *dev);
```

The next two functions, unregister_netdev and unregister_netdevice, remove the device dev from the list of devices. The first of these functions, unregister_netdev, is the "safe" version. In this routine, we take the rt_netlink semaphore and then call the internal routine unregister_netdevice.

```
void        unregister_netdev(struct net_device *dev);
```

The next function also unregisters a net_device. In this function, we require the caller to take the rtnl semaphore before invocation.

```
int unregister_netdevice(struct net_device *dev)
{
    struct net_device *d, **dp;

    BUG_ON(dev_boot_phase);
    ASSERT_RTNL();
```

We also protect against being called with a device that has never been registered.

```
    if (dev->reg_state == NETREG_UNINITIALIZED) {
        printk(KERN_DEBUG "unregister_netdevice: device %s/%p never "
               "was registered\ n", dev->name, dev);
        return -ENODEV;
    }

    BUG_ON(dev->reg_state != NETREG_REGISTERED);
```

Next, we check for IFF_UP in the flags of the network device structure to see if the device is running. If the device is running, we call the network interface generic dev_close function to stop the device.

```
    if (dev->flags & IFF_UP)
        dev_close(dev);
```

Next, we remove the pointer to the net_device structure from the linked list of devices, and shut down the device's queues for the particular queuing discipline in effect by calling dev_shutdown.

```
    for (dp = &dev_base; (d = *dp) != NULL; dp = &d->next) {
        if (d == dev) {
            write_lock_bh(&dev_base_lock);
            *dp = d->next;
            write_unlock_bh(&dev_base_lock);
            break;
        }
    }
    if (!d) {
        printk(KERN_ERR "unregister net_device: ‘%s’ not found\ n",
               dev->name);
        return -ENODEV;
    }
    dev->reg_state = NETREG_UNREGISTERING;
    synchronize_net();
#ifdef CONFIG_NET_FASTROUTE
    dev_clear_fastroute(dev);
#endif
    dev_shutdown(dev);
```

We send a notification message to any interested protocols that this device is about to be destroyed by calling the notifier_call_chain, covered in Section 4.10.

```
    notifier_call_chain(&netdev_chain, NETDEV_UNREGISTER, dev);
```

We flush the list of multicast addresses from the device.

```
    dev_mc_discard(dev);
    if (dev->uninit)
        dev->uninit(dev);
```

This is a check of the notifier chain. It should have detached us from the master device in the notifier group.

```
    BUG_TRAP(!dev->master);
```

```
    free_divert_blk(dev);
```

In Linux 2.6, device registration is asynchronous. Work on device registration and unregistration can be deferred if a device is busy.

```
    net_set_todo(dev);
```

Finally, we check the reference count in the refcnt field of the net_device structure by calling dev_put. If it is one or higher, a reference to the device must be active, and we can decrement the reference count.

```
    dev_put(dev);
    return 0;
}
```

To complete this section on network device registration, we will list a few functions that Linux provides to retrieve a device. Each of these functions returns a net_device structure based on various criteria such as type and hardware address, flags, or ifindex.

```
struct net_device       *dev_getbyhwaddr(unsigned short type, char *hwaddr);
struct net_device       *__dev_getfirstbyhwtype(unsigned short type);
struct net_device       *dev_getfirstbyhwtype(unsigned short type);
struct net_device       *dev_get_by_flags(unsigned short flags,
                                          unsigned short mask);
struct net_device       *__dev_get_by_flags(unsigned short flags,
struct net_device       *dev_get_by_index(int ifindex);
```

There are several functions provided to allocate a device structure or merely a device name string. The first function, dev_alloc_name, checks for a valid format in name. It appends a digit 1 to 99 to the name and returns the number appended to the name.

```
int dev_alloc_name(struct net_device *dev, const char *name);
```

The next function, alloc_etherdev, allocates a complete net_device structure specifically for an Ethernet device.

```
struct net_device *alloc_etherdev(int sizeof_priv);
```

The last function in this group, alloc_netdev, is generally not called directly. Instead, it is called by one of the functions that are specific to the device type, such as alloc_etherdev in the preceding code. Alloc_netdev allocates the private data area and the net device structure. It also initializes the name field in the net_device structure to the base string for the name such as "eth".

```
struct net_device *alloc_netdev(int sizeof_priv, const char *mask,
                                void(*setup)(struct net_device *));
```

Finally, a function, netdev_boot_setup, is provided to all driver writers to set fields in the net_device structure from boot time parameters.

```
int __init netdev_boot_setup(char *str);
```

4.7 Network Interface Driver Service Functions

As discussed in earlier sections, the network interface driver service functions have been set in the dev structure during driver initialization. Each of the service functions is associated with generic versions implemented in the file *linux/net/core/dev.c*, but are often over-ridden by the network interface driver's specific function or by a function for a particular network interface device type. The service functions are not called directly by application code or by any protocols in the TCP/IP suite. Instead, they are called by de-referencing the corresponding pointers in the net device structure.

Most of the functions affect the state of the network interface, whether it is up, down, or in some other state. The state is maintained in the flags field in the net device structure, and most of the interface service functions will change the flags. The values for the network interface flags are shown in Table 4.4.

Table 4.4: Network Interface Flags

Flag	**Value**	**Meaning**
IFF_UP	0x1	The interface is up.
IFF_BROADCAST	0x2	The broadcast address is valid.
IFF_DEBUG	0x4	Turns on debugging.
IFF_LOOPBACK	0x8	The interface is a loopback interface.
IFF_POINTOPOINT	0x10	Interface is a point-to-point link.
IFF_NOTRAILERS	0x20	Avoid the use of trailers.
IFF_RUNNING	0x40	The interface is running and resources allocated.
IFF_NOARP	0x80	The ARP protocol is not supported on this interface.
IFF_PROMISC	0x100	The interface is in promiscuous mode. It receives all packets.
IFF_ALLMULTI	0x200	The interface receives all multicast packets.
IFF_MASTER	0x400	Master of a load balancer.
IFF_SLAVE	0x800	Slave of a load balancer.
IFF_MULTICAST	0x1000	The interface supports multicast.
IFF_VOLATILE	(IFF_LOOPBACK|IFF_POINTOPOINT|IFF_BROADCAST| IFF_MASTER|IFF_SLAVE|IFF_RUNNING)	
IFF_PORTSEL	0x2000	Media type can be set on this interface.
IFF_AUTOMEDIA	0x4000	The auto media select is active.
IFF_DYNAMIC	0x8000	Interface is a dial-up device with changing addresses.

In this section, both the driver's specific function and the generic function are discussed. In most cases, we discuss the generic function because much of the work that actually glues the driver to the TCP/IP stack is done in the generic function.

In this section, the more common network interface service functions are described. The functions described in this section must be implemented by all device drivers.

4.7.1 Open Network Interface Service Function

The driver's open function is called through the open field in the net device structure.

```
int (*open) (struct net_device *dev);
```

Open is called by the generic dev_open function in *linux/net/core/dev.c*.

```
int dev_open(struct net_device *dev);
```

First, dev_open checks to see if the device has already been activated by checking for IFF_UP in the flags field of the net_device structure, and if the driver is already up, we simply return a zero. Next, it checks to see if the physical device is present by calling netif_device_present, which checks the link state bits in the state field of the network device structure. If all this succeeds, dev_open calls the driver through the open field in the net_device structure.

Most drivers use the open function to initialize their internal data structures prior to accepting and transmitting packets. These structures may include the internal queues, watchdog timers, and lists of internal buffers. Next, the driver generally starts up the receive queue by calling netif_start_queue, defined in *linux/include/linux/netdevice.h*, which starts the queue by clearing the __LINK_STATE_XOFF in the state field of the net_device structure. The states are listed in Table 4.1 and are used by the queuing layer to control the transmit queues for the device. See Section 4.8 for a description of how the packet queuing layer works. Right up to the point where the queuing is started, the driver can change the device's queuing discipline. Chapter 6 has more detail about Linux's capability to work with multiple queuing disciplines for packet transmission queues.

After the driver's open returns, dev_open does a little bit more housekeeping. If the driver's open function encounters an error, dev_open returns ENODEV right away. If everything is OK, the flags field is set to IFF_UP and the state field is set to LINK_STATE_START to indicate that the network link is active and ready to receive packets. Next, dev_open calls the dev_mc_upload to set up the list of multicast addresses for this device. Finally, dev_open calls dev_activate, which sets up a default queuing discipline for the device, typically pfifo_fast for hardware devices and none for pseudo or software devices.

4.7.2 Set_Multicast_list Network Interface Service Function

The set_multicast_list driver service function initializes the list of multicast addresses for the interface.

```
Void    (*set_multicast_list)(struct net_device *dev);
```

Device multicast addresses are contained in a generic structure, dev_mc_list, which allows the interface to support one or more link layer multicast addresses.

```
struct dev_mc_list
{
    struct        dev_mc_list    *next;
    __u8          dmi_addr[MAX_ADDR_LEN];
    unsigned char dmi_addrlen;
    int           dmi_users;
    int           dmi_gusers;
} ;
```

This structure is internal to the multicast maintenance functions and isn't manipulated directly by the device driver or any of the protocols. Utility functions are provided to manipulate the device multicast list. The list maintains use counts of multicast addresses per interface. Chapter 9 provides more detailed information on how Linux does multicasting routing for IPv4.

4.7.3 Hard_start_xmit Network Interface Service Function

The hard_start_xmit network interface service function starts the transmission of an individual packet or queue of packets.

```
int (*hard_start_xmit) (struct sk_buff *skb, struct net_device *dev);
```

This function is called from the network queuing layer when a packet is ready for transmission. Section 4.8 has more information on the packet queuing layer. The first thing the driver must do in hard_start_xmit is ensure that there are hardware resources available for transmitting the packet and there is a sufficient number of available buffers. If buffers are available, it enables transmission for the hardware's PHY chip. For a typical Ethernet driver, we would simply enable the Ethernet chip for transmission and return. If there are no hardware buffers available, the packet is re-queued. If the driver is a pseudo-driver, it would have no transmission hardware and no separate transmission queue within the driver, so the buffer can be removed from the queue for processing immediately. If the packet can't be transmitted for some reason, the queuing layer is informed by calling netif_stop_queue.

4.7.4 Uninit Network Interface Service Function

The uninit network interface service function is responsible for any device-specific cleanup when the device is unregistered.

```
Void    (*uninit)(struct net_device *dev);
```

Uninit is called from the unregister_netdevice function in *linux/net/core/dev.c*, covered earlier. Generally, only pseudo-device drivers that have protocol functionality will implement this function. In our example, the e100 driver, we do not implement uninit; generally, Ethernet or similar device drivers will not need to provide this network service function. In addition, if the device driver is a new style device that implements the destructor service function, it could be relied on to do some of the cleanup.

4.7.5 Stop Network Interface Service Function

The stop network interface service function could be thought of as the close function for the device and is called when the kernel wants to tell the driver to terminate packet transmission.

```
int (*stop)(struct net_device *dev);
```

In our example, the e100 driver, the stop service function is set to e100_close. This function turns off transmission interrupts (if applicable) and cleans up any open internal transmission buffers. An internal close flag (in the private part of the driver's data structure) is used in many drivers to keep track of its open or closed state. Generally, this function will check to see if the driver is up, the IFF_UP flag in the flags field of net_device. If so, the queuing layer is told to stop queuing packets by calling netif_stop_queue.

4.7.6 Change_mtu Network Interface Service Function

The change_mtu network interface service function is to change the Maximum Transmission Unit (MTU) of a device.

```
int (*change_mtu)(struct net_device *dev, int new_mtu);
```

Since the MTU is generally fixed for a particular device type, this function is rarely used. It is not necessary that this field is initialized in net_device driver's initialization function.

4.7.7 Get_stats—Get Statistics Network Interface Service Function

Get_stats returns a pointer to the network device statistics.

```
struct net_device_stats* (*get_stats)(struct net_device *dev);
```

Generally, network interface drivers will keep the network device statistics in the private part of the network interface driver's data structure, so this function is provided to retrieve them. The net_device_stats structure is defined in file *linux/include/linux/net*[*device.h*](). This structure contains counters for the number of packets received and transmitted, the number of bytes received and transmitted, the number of dropped packets, and other statistics.

4.7.8 Do_ioctl—IO Control Network Interface Service Function

Do_ioctl implements any device-specific socket IO control (ioctl) functions.

```
int (*do_ioctl)(struct net_device *dev, struct ifreq *ifr, int cmd);
```

Not all network interface drivers will implement do_ioctl. In most cases, socket-layer ioctl calls are handled by the generic device layer function, dev_ioctl in *linux/net/core/*[*dev.c*](). When dev_ioctl can't handle the request and the do_ioctl service function entry is defined for the specific device, the driver's do_ioctl entry point is called. It is not required that the network interface driver initialize this function pointer. The socket IO controls are defined in the file *linux/include/linux/*[*sockios.h*]().

4.7.9 Destructor Function

The Destructor isn't technically a service function.

```
void (*destructor)(struct net_device *dev);
```

We include it in this section because as is the case with the service functions, destructor is an entry point in the driver called through the net_device structure. Destructor does any final cleanup, including de-allocating the net_device structure. Not all network interface drivers need to implement this function. In many cases, the destructor field of net_device points to free_netdev defined in *linux/net/core/dev.c*.

4.8 Receiving Packets

As is the case with any hardware device driver in other operating systems, the first step in packet reception occurs when the device responds to an interrupt from the network interface hardware. With Ethernet drivers, and our example of e100, when DMA of an incoming packet is complete, the network interface driver's interrupt handler is called. In the interrupt handler, we first detect the cause of the interrupt. We determine if the interrupt is valid, and if so, whether it is a transmit or a receive interrupt. If we detect a receive interrupt, we know that a received packet is available for processing so we can begin to perform the steps necessary for packet reception. One of the first things we must do is gather the buffer containing the raw received packet into a socket buffer or sk_buff. Most efficient drivers will avoid copying the data at this step. In Linux, the socket buffers are used to contain network data packets, and they can be set up to point directly to the DMA space. Chapter 7 covers socket buffers in detail. Generally, network interface drivers maintain a list of sk_buffs in their private data structure, and once the interrupt indicates that input DMA is complete, we can place the socket buffer containing the new packet on a queue of packets ready for processing by the protocol's input function.

For efficiency, we generally want network interface drivers to return from the interrupt as soon as possible. While we are processing an interrupt, all other processes are suspended; therefore, we want to do as little as possible while we are in the interrupt service routine. Therefore, we queue up the packet for further processing in the softirq context to offload as much processing as possible from the interrupt service routine. Table 4.5 shows functions used to control the device state shown earlier in Table 4.1. Each of these inline functions are used by network interface drives to control the queues.

Table 4.5: Network Queuing Layer Functions

Function from File linux/include/linux/ netdevice.h	**Arguments**	**Returns**	**Purpose**
netif_start_queue	struct net_device *	void	Starts the queue.
netif_receive_skb	struct sk_buff *	int	Packet receive function. Netif_receive_skb is called in the case of congestion to try to

Table 4.5: Network Queuing Layer Functions

Function from File linux/include/linux/ netdevice.h	**Arguments**	**Returns**	**Purpose**
			process backlog.
netif_schedule	struct net_device *	void	This function schedules output queue processing.
netif_wake_queue	struct net_device *	void	Restarts transmission.
netif_stop_queue	struct net_device *	void	Turns off transmission of output queue.
netif_queue_stopped	struct net_device *	int	Checks state of transmission link to see whether transmission is turned off.
netif_running	struct net_device *	int	Checks to see if transmission is in the start state.

In most network interface drivers, the interrupt handler must first do the processing associated with packet reception. It does some internal housekeeping for mapping and unmapping device space, maintaining device state, setting up for the next DMA transfer, and ensuring that a sufficient number of socket buffers are available for subsequent DMA transfers. Once these internal functions are complete, the interrupt handler sets the packet type in the socket buffer by setting the protocol field to the type in the Ethernet packet header. It also adjusts the socket buffer to make sure there is sufficient tail room for the packet size by calling skb_put. Refer to Chapter 7 for a description of skb_put and the other socket buffer utility functions. The netif_rx function declared in file *linux/include/linux/netdevice.h* is called by the ISR to invoke the input side of packet processing and queue up the packet for processing by the packet receive softirq, NET_RX_SOFTIRQ.

```
int netif_rx(struct sk_buff *skb);
```

Chapter 6 includes a detailed description of the packet queuing layer and the softirq capability. The netif_rx function returns a value indicating the amount of network congestion detected by the queuing layer or whether the packet was dropped altogether. Table 4.6 shows the return values for netif_rx.

Table 4.6: Queuing Layer netif_rx Return Values

Value	**Meaning**
NET_RX_SUCCESS	No congestion
NET_RX_CN_LOW	Low congestion
NET_RX_CN_MOD	Moderate congestion
NET_RX_CN_HIGH	High congestion
NET_RX_DROP	Packet was dropped

Another function, netif_rx_ni, in file *linux/include/linux/netdevice.h*, is provided to queue up input packets from a noninterrupt context. This function reschedules the softirq after queuing up the incoming packet. It also returns the values described in Table 4.6.

```
static inline int netif_rx_ni(struct sk_buff *skb)
{
    int err = netif_rx(skb);
    if (softirq_pending(smp_processor_id()))
        do_softirq();
    return err;
}
```

The next function, netif_rx, is the main function called from interrupt service routines in network interface drivers. It is defined in file *linux/net/core/dev.c*. It is passed a pointer to a socket buffer holding the new received packet.

```
int netif_rx(struct sk_buff *skb)
{
    int                this_cpu;
    struct softnet_data *queue;
    unsigned long       flags;
```

First, we set the stamp filed in the sk_buff (if there is none there already) with the timestamp of when the packet was received.

```
    if (skb->stamp.tv_sec == 0)
        do_gettimeofday(&skb->stamp);
```

The code in this function is written so the execution path is shortest when the CPU is congested. We get a pointer to the queue from the current softnet_data structure.

```
    local_irq_save(flags);
    this_cpu = smp_processor_id();
    queue = &__get_cpu_var(softnet_data);
    __get_cpu_var(netdev_rx_stat).total++;
```

We check for backlog by seeing if the input packet queue is longer than the constant value set in the netdev_max_backlog global variable defined in *dev.c*.

```
    __get_cpu_var(netdev_rx_stat).total++;
    if (queue->input_pkt_queue.qlen <= netdev_max_backlog) {
```

We also check to see if the throttle field is set in the softnet_data structure. If there is no congestion indication, we call netif_rx_schedule.

```
        if (queue->input_pkt_queue.qlen) {
            if (queue->throttle)
                goto drop;
enqueue:
            dev_hold(skb->dev);
            __skb_queue_tail(&queue->input_pkt_queue, skb);
#ifndef OFFLINE_SAMPLE
```

```
            get_sample_stats(this_cpu);
#endif
            local_irq_restore(flags);
            return queue->cng_level;
        }
```

If flow control is configured, we attempt to do it.

```
        if (queue->throttle) {
            queue->throttle = 0;
#ifdef CONFIG_NET_HW_FLOWCONTROL
            if (atomic_dec_and_test(&netdev_dropping))
                netdev_wakeup();
#endif
        }
```

We call netif_rx_schedule to add the interface to the tail of the poll list. This effectively reschedules the interface for later processing when the softirq executes.

```
        netif_rx_schedule(&queue->backlog_dev);
        goto enqueue;
    }
    if (!queue->throttle) {
        queue->throttle = 1;
        __get_cpu_var(netdev_rx_stat).throttled++;
#ifdef CONFIG_NET_HW_FLOWCONTROL
        atomic_inc(&netdev_dropping);
#endif
    }
```

Here we drop the packet; there is not much to do—increment the statistics, free the skb, and indicate to the caller the congestion indication, which of course is NET_RX_DROP.

```
drop:
    __get_cpu_var(netdev_rx_stat).dropped++;
    local_irq_restore(flags);

    kfree_skb(skb);
    return NET_RX_DROP;
}
```

We can't talk about the queuing layer input side without including softnet_data, a data structure defined in file *linux/include/linux/netdevice.h*. Starting with Linux version 2.4, this structure includes a copy of a pseudo net device structure called blog_dev, otherwise known as the backlog device.

```
struct softnet_data
{
    int                 throttle;
    int                 cng_level;
    int                 avg_blog;
    struct sk_buff_head input_pkt_queue;
    struct list_head    poll_list;
    struct net_device   *output_queue;
```

```
    struct sk_buff      *completion_queue;

    struct net_device   backlog_dev;
} ;
```

Backlog_dev is used by the packet queuing layer to store the packet queues for most nonpolling network interface drivers. The blog_dev device is used instead of the "real" net_device structure to hold the queues, but the actual device is still used to keep track of the network interface from which the packet arrived as the packet is processed by the upper layer protocols. As we have seen, there are many fields defined in the net device structure. The only fields used in blog_dev are poll_list, quota, state, weight, poll, and refcnt.

4.9 Transmitting Packets

Packet transmission is controlled by the upper layers, not by the network interface driver. To understand what happens when a packet is transmitted, we examine what happens between the time the network layer protocol such as IP decides to transmit a packet and the time the driver's transmit function is started. Earlier, we described the hard_start_xmit network service function, which does the actual packet transmission. That function is actually called from the packet queuing layer when there is one or more packets in a socket buffer ready to transmit. In most drivers, when it is called from the queuing layer, hard_start_xmit will put the sk_buff on a local queue in the driver's private data structure and enable the transmit available interrupt. Once the interrupt is enabled, the actual transmission will happen when the interrupt service routine executes. Earlier, we covered the hard_start_xmit and how it is implemented in a sample Ethernet network interface driver.

There is a complex body of code involved with queuing the packet to the driver for transmission. The functional details of this series of steps are described in detail in Chapter 6. As we shall see in Chapter 6, Linux allows multiple queuing disciplines to be used with packet transmission. We will see how the queuing disciplines are loaded and registered with the kernel. However, in this section we will describe the process of transmitting packets from the protocol's output function using the default scheduler and default queuing discipline. The default queuing discipline used for hardware drivers is pfifo_fast, and for software or pseudo drivers, it is noop. Both of these are implemented in the file *linux/net/sched/sch_generic.c*.

The queuing layer function involved in transmitting packets is dev_queue_xmit defined in file *linux/net/core/dev.c*. It is independent of the network layer protocol and independent from any particular queuing discipline used for a particular transmission path.

```
int dev_queue_xmit(struct sk_buff *skb)
{
    struct net_device   *dev = skb->dev;
    struct Qdisc        *q;
    int                 rc = -ENOMEM;
```

First, we check the features flag, NETIF_F_FRAGLIST, to see if the skb is split into a list of fragments. In addition, we check for the flag NETIF_F_SG to see if the device supports scatter gather IO. *Scatter gather IO* is the term for doing DMA to or from noncontiguous memory regions. In Chapter 7, we will see how a segmented socket buffers consists of a single IP header

and a list of fragments that all use the same shared header info. If the packet is fragmented but the device does not have the scatter gather capability, then the packet must be linearized, which means that the IP header and data portion of the packet is gathered into a single linear skb.

```
if (skb_shinfo(skb)->frag_list &&
    !(dev->features & NETIF_F_FRAGLIST) &&
    __skb_linearize(skb, GFP_ATOMIC))
    goto out_kfree_skb;
```

Next, dev_queue_xmit checks the features flags for NETIF_F_HW_CSUM to see if the device has the capability of calculating a checksum in hardware. In addition, it checks to see if the packet already has an IP checksum by looking at the ip_summed field in the socket buffer. If the device can't calculate the checksum in hardware and the packet doesn't already have a checksum, we call skb_checksum_help to calculate the checksum in software.

```
if (skb->ip_summed == CHECKSUM_HW &&
    (!(dev->features & (NETIF_F_HW_CSUM | NETIF_F_NO_CSUM)) &&
    (!(dev->features & NETIF_F_IP_CSUM) ||
       skb->protocol != htons(ETH_P_IP)))) {
    if ((skb = skb_checksum_help(skb)) == NULL)
        goto out;
}
```

At this point, dev_queue_xmit may re-queue the packet or it may send it directly to the driver depending on the queue discipline that is in effect for this device driver, the state of the queue, and the traffic class scheduler in use. We get the device queue discipline from the qdisc field of the net_device structure. If we have a queue, we put the packet on the queue.

```
spin_lock_bh(&dev->queue_lock);
q = dev->qdisc;
if (q->enqueue) {
    rc = q->enqueue(skb, q);

    qdisc_run(dev);

    spin_unlock_bh(&dev->queue_lock);
    rc = rc == NET_XMIT_BYPASS ? NET_XMIT_SUCCESS : rc;
    goto out;
}
```

If this device has no queue, we call the dev_queue_xmit_nit to send the packet.

```
if (dev->flags & IFF_UP) {
    int cpu = smp_processor_id();

    if (dev->xmit_lock_owner != cpu) {
```

The spin_lock effectively does a preemption here; however, we want to disable pre-emption at this point.

```
        preempt_disable();
        spin_unlock(&dev->queue_lock);
        spin_lock(&dev->xmit_lock);
```

```
            dev->xmit_lock_owner = cpu;
            preempt_enable();

            if (!netif_queue_stopped(dev)) {
```

Netdev_nit will send the outgoing packet to all internal listeners or "network taps."

```
                if (netdev_nit)
                    dev_queue_xmit_nit(skb, dev);
                rc = 0;
```

Here is where we call the network interface driver's transmit routine.

```
                if (!dev->hard_start_xmit(skb, dev)) {
                    dev->xmit_lock_owner = -1;
                    spin_unlock_bh(&dev->xmit_lock);
                    goto out;
                }
            }
```

Finally, we check for rate limiting and packet recursion.

```
            dev->xmit_lock_owner = -1;
            spin_unlock_bh(&dev->xmit_lock);
            if (net_ratelimit())
                printk(KERN_CRIT "Virtual device %s asks to "
                       "queue packet!\ n", dev->name);
            goto out_enetdown;
        }  else {
            if (net_ratelimit())
                printk(KERN_CRIT "Dead loop on virtual device "
                       "%s, fix it urgently!\ n", dev->name);
        }
    }
    spin_unlock_bh(&dev->queue_lock);
out_enetdown:
    rc = -ENETDOWN;
out_kfree_skb:
    kfree_skb(skb);
out:
    return rc;
}
```

See Chapter 6 for more detail about queue disciplines, and Chapter 9 for how the queue disciplines are used for routing and traffic class based scheduling.

4.10 Notifier Chains and Network Interface Device Status Notification

Linux provides a mechanism for device status change notification called notifier chains. It is based on a generic notifier facility that can be used by modules anywhere in the kernel for various purposes. In the TCP/IP stack, the notifier chains are used to pass changes in network device status or other events to any protocols, modules, or devices that register themselves with the facility. The notifier chains are for passing changes in status to any function that is registered

with the notifier facility. Table 4.7 shows the pre-declared notifier events that are used currently in the Linux kernel. These values are declared in file *linux/include/linux/notifier.h*. The notifier chain is actually a very simple construct and isn't much more than a linked list of functions to call for event notification.

Table 4.7: Pre-Declared Net Device Notification Events

Name	**Value**	**Purpose**
NETDEV_UP	0x0001	Network device up notifier.
NETDEV_DOWN	0x0002	Network device down notifier.
NETDEV_REBOOT	0x0003	Tells a protocol that a network interface detected a hardware crash and restarted.
NETDEV_CHANGE	0x0004	Notification of device state change.
NETDEV_REGISTER	0x0005	Notification of a network device registration event.
NETDEV_UNREGISTER	0x0006	Notification of a network device un-registration event.
NETDEV_CHANGEMTU	0x0007	Network device MTU change notifier.
NETDEV_CHANGEADDR	0x0008	Network device address change notifier.
NETDEV_GOING_DOWN	0x0009	Notification that a network device is going down.
NETDEV_CHANGENAME	0x000A	Notify that a network device has changed its name.

Each location in the linked list is defined by an instance of the notifier_block structure. This structure is generally initialized at compile time by the code that wants to use the notification facility.

```
struct notifier_block
{
```

The first field in this structure, notifier_call, is the event notification function. This field is initialized statically.

```
    int (*notifier_call)(struct notifier_block *self, unsigned long,
void *);
```

Next is not allocated statically. A module that expects to receive event notification must declare an instance of notifier_block and set the next field to NULL and is filled in later when the block is registered.

```
    struct notifier_block *next;
    int priority;
} ;
```

4.10.1 Generic Event Notification Functions

Linux provides a set of generic functions for event notification. The first of these, notifier_chain_register, registers a notifier_block with the event notification facility.

```
int notifier_chain_register(struct notifier_block **list,
struct notifier_block *n);
```

The parameter n is set to point to the notifier_block, and list points to the notifier event chain.

To unregister with the event notification facility, the notifier unregistration function is called.

```
int notifier_chain_unregister(struct notifier_block **nl,
struct notifier_block *n);
```

If the event notification facility is used in a module, the notifier_chain_unregister must be called before the module is unloaded. There is no protection of the notification call chains. Once registered, the function pointed to by notifier_call cannot be reused or altered until after the unregistration is complete.

To pass an event into the notification call chain, the function notifier_call_chain is called with a pointer to the notifier_block list, n an event value, val, and an optional generic argument, v.

```
int notifier_call_chain(struct notifier_block **n, unsigned long val,
void *v);
```

4.10.2 Netdev_chain Notifier Call Chain

In the previous section, we looked at the Linux notification event facility. There is also a specific event notification facility for network interface devices called the netdev_chain in file *linux/net/core/dev.c*. The netdev_chain is used widely throughout the Linux TCP/IP protocol family, and its main purpose is to inform protocols when network devices are brought up and down. Through this facility, recipients are notified of a number of predefined events related to network interface device status, shown in Table 4.7. The functions provided to register and unregister with the netdev_chain are also implemented in file *linux/net/core/dev.c* and have prototypes defined in file *linux/include/linux/netdevice.h*. As is the case with the generic facility, a user registers a notifier_block, and neither the block nor the function in it can be reused until the function is unregistered. To use this facility, we create a notifier_block instance usually at compile time. The block contains a pointer to the function that will be called when the event occurs.

To register with the call chain, we call the registration function register_netdevice_notifier with a pointer to the notifier_block.

```
int register_netdevice_notifier(struct notifier_block *nb);
```

The caller to this function doesn't pick or choose which events it will see.

A notification function will be called through the notifier_call field in nb for all the event types in Table 4.7.

```
int (*notifier_call)(struct notifier_block *self, unsigned long, void *);
```

This function should explicitly check for the types of events it wants to process. The first argument, self, will point back to the notifier_block; the second argument will be one of the event types from Table 4.7; and the third argument will be a pointer to a net_device structure indicating the device that caused the event.

To unregister with the net device notifier chain, we call unregister_netdevice_notifier to remove the notifier_block from the call chain.

```
int unregister_netdevice_notifier(struct notifier_block *nb);
```

4.11 Summary

In this chapter, we provided a background on Linux network interfaces to provide a foundation for more detailed discussion in later chapters on the internals of Linux socket buffers and the network and transport protocols. We covered the network interface drivers and Linux TCP/IP features that are directly related to network interfaces, and discussed the interface between network interface drivers and the queuing layer. In addition, the driver's network interface driver service functions were discussed. We covered the registration and de-registration process for network interfaces, and the netdev_chain event notification facility. We explained the net_device structure along with a description of all the important fields. The reader should leave this chapter with a good understanding of how Linux network interface drivers work and how they interface to the TCP/IP stack.

Chapter 5: Linux Sockets

The socket interface was originally developed as part of the BSD operating system. Sockets provide a standard protocol-independent interface between the application-level programs and the TCP/IP stack. As discussed in Chapter 1, "Introduction," the OSI model serves us well as the framework to explain networking protocol stacks. It defines all seven layers, including the three layers above the transport layer—session, presentation, and application. As we know, TCP/IP does not define these three upper layers. As a matter of fact, it does not define anything above the transport layer. From the viewpoint of TCP/IP, everything above the transport layer is part of the application. Linux is similar to traditional Unix in the sense that the TCP/IP stack lives in the kernel, sharing memory space with the rest of the kernel. All the network functions performed above the transport layer are done in the user application space. Linux provides an API and sockets, which are compatible with Unix and many other systems. Applications use this API to access the networking facilities in the kernel.

5.1 Introduction

The original intent of Linux was to create an Operating System (OS) that is functionally compatible with the proprietary versions of Unix that were popular at the time Linux was originally developed. The socket API is the best known networking interface for Unix application and network programming. Linux has not disappointed us in that it has provided a socket layer that is functionally identical to traditional Unix. Almost every type of TCP/IP-based networking application has been successfully ported to Linux over many years and includes applications used in the smallest embedded system to the largest servers. Many excellent books on Unix network programming do a great job of explaining the socket API and Unix network layer programming; see [STEV98] for an excellent book. In this book we will not duplicate earlier efforts by providing elaborate explanations of application layer protocols. Instead, our intent is to explain the underlying structure of Linux TCP/IP. Therefore, although the socket API will be discussed, the emphasis is on the underlying infrastructure.

The Linux socket API conforms to all applicable standards, and application layer protocols are entirely portable to Linux, from other flavors of Unix and many other OSs as well. However, the underlying infrastructure of the socket layer implementation is unique to Linux. In this chapter we discuss netlink sockets and other ways that an application programmer can use sockets to interact with the protocols and layers in the TCP/IP stack. We discuss the definition of a socket, a detailed discussion of the sock structure, the socket API, and the socket call mapping for file operations. In addition, we cover how sockets can be used with a new protocol and how the netlink mechanism allows applications to control the operations of the internal protocols in TCP/IP.

5.2 What is a Socket?

One definition of the socket interface is that it is the interface between the transport layer protocols in the TCP/IP stack and all protocols above. However, the socket interface is also the interface between the kernel and the application layer for all network programming functions. All data and control functions for the TCP/IP stack pass through the socket interface. As we saw

in the introductory chapters in this book, the TCP/IP stack itself does not include any protocols above the transport layer. Instead, Linux and most other operating systems provide a standard interface called *sockets* for all protocols above the transport layer. The socket interface is really TCP/IP's window on the world. In most modern systems incorporating TCP/IP—and this includes Linux—the socket interface is the only way that applications make use of the TCP/IP suite of protocols.

Sockets have three fundamental purposes. They are used to transfer data, manage connections for TCP, and control or tune the operation of the TCP/IP stack. The socket interface is an elegant and simple design. This is probably the primary factor leading to the wide acceptance of the TCP/IP protocol stack by application programmers over many years. Sockets are generic—they have the capability of working with protocol suites other than TCP/IP, including Linux's internal process-to-process communication, AF_UNIX. The protocol family types supported by Linux sockets are listed in Table 5.1. The complete list of official protocol families is part of the assigned numbers database that is currently maintained by the Internet Assigned Numbers Authority (IANA) [IAPROT03].

Table 5.1: Supported Protocol and Address Families

Name	Value	Protocol
AF_UNSPEC	0	The address family is unspecified.
AF_UNIX	1	Unix domain sockets.
AF_LOCAL	1	The same as AF_UNIX address family.
AF_INET	2	Internet, the TCP/IP protocol family.
AF_AX25	3	Amateur radio AX.25 protocol.
AF_IPX	4	IPX protocol sockets.
AF_APPLETALK	5	AppleTalk DDP sockets.
AF_NETROM	6	NET/ROM sockets.
AF_BRIDGE	7	Multiprotocol bridge sockets.
AF_ATMPVC	8	ATM PVX sockets.
AF_X25	9	X.25 sockets.
AF_INET6	10	IP version 6.
AF_ROSE	11	Amateur radio X.25 PLP.
AF_DECnet	12	Reserved for DECnet sockets.
AF_NETBEUI	13	Reserved for 802.2LLC project.
AF_SECURITY	14	Security callback pseudo address family.
AF_KEY	15	Sockets for PF_KEY key management API.
AF_NETLINK	16	Netlink protocol sockets. Message passing between applications and kernel. See Section 5.9.
AF_ROUTE	16	Alias to emulate 4.4BSD routing sockets. Same as AF_NETLINK in Linux.

Table 5.1: Supported Protocol and Address Families

Name	Value	Protocol
AF_PACKET	17	Packet family sockets.
AF_ASH	18	Ash sockets.
AF_ECONET	19	Acorn Econet sockets.
AF_ATMSVC	20	ATM SVC sockets.
AF_SNA	22	Linux SNA sockets.
AF_IRDA	23	IRDA sockets.
AF_PPPOX	24	PPPoX sockets.
AF_WANPIPE	25	Wanpipe API sockets.
AF_LLC	26	Linux LLC sockets.
AF_BLUETOOTH	31	For Bluetooth sockets.
AF_MAX	32	Currently, only 32 address families are supported.

The socket API actually has two parts. It consists of a set of functions specifically for a network. It also contains a way of mapping standard Unix I/O operations so application programmers using TCP/IP to send and receive data can use the same calls that are commonly used for file I/O. Instead of open, the socket function is used to open a socket. However, once the socket is open, generic I/O calls such as read and write can be used to move data through the open socket.

5.3 Socket, sock, and Other Data Structures for Managing Sockets

Sometimes the choice of names for data structures and functions can be confusing, and this is definitely the case when we discuss sockets in Linux. As with other operating systems, Linux uses similar names or terms to describe functionally different data structures. For example, several data structures for sockets can be confused with each other. In Linux, the three different data structures each have the letters "sock" in them. The first is the socket buffer defined in *linux/include/linux/sk_buff.h*. Socket buffers are structures to hold packet data that may or may not be created at a socket interface and are covered in detail in Chapter 7. In the Linux source code, socket buffers are often referred to by the variable sk6.

```
struct sk_buff *skb;
```

The next two data structures are covered in this chapter. The second data structure that we will discuss is the socket structure defined in *linux/include/linux/net.h*. The socket structure is not specific to TCP/IP. Instead, it is a generic structure used primarily within the socket layer to keep track of each open connection and as a vehicle to pass open sockets to and from the socket layer. Generally, each instance of a socket structure corresponds to an open socket that was open with the socket call. Sockets are also implicitly referenced in the application code by the file descriptor returned by socket. This socket structure is usually referenced by a variable called sock.

```
struct socket *sock;
```

The third data structure is another one we will discuss in detail in this chapter. It is called the sock structure and is defined in the file *linux/include/net/sock.h*. It is a more complex structure used to keep state information about open connections. It is accessed throughout the TCP/IP protocol but mostly within the TCP protocol. It is usually referenced through a variable called sk.

```
struct sock *sk;
```

5.3.1 The Sock Structure

In earlier versions of the kernel, the sock structure was much more complex. However, in 2.6, it has been greatly simplified in two ways. The structure is preceded with a common part that is generic to all protocol families. The other way it is different is that in Linux 2.6, instances of the sock structure are allocated from protocol-specific slab caches instead of a generic cache. In addition, following the structure there is an IPv4 and IPv6 specific part that contains the prot_info structure for each of the member protocols in the protocol family. This part is discussed in Section 5.11.1 as part of the discussion of socket creation.

The first part of the sock structure is kept in a structure called sock_common.

```
struct sock_common {
```

The first field in skc_family contains the network address family, such as AF_INET for IPv4. Skc_state is the connection state, and skc_reuse holds the value of the SO_REUSEADDR socket option.

```
    unsigned short            skc_family;
    volatile unsigned char    skc_state;
    unsigned char             skc_reuse;
```

The next field, skc_bound_dev_if holds the index for the bound network interface device.

```
    int                       skc_bound_dev_if;
```

The next two fields hold hash linkage for the protocol lookup tables. Finally, skc_refcnt is the reference count for this socket.

```
    struct hlist_node         skc_node;
    struct hlist_node         skc_bind_node;
    atomic_t                  skc_refcnt;
} ;
```

Now we will look at the sock structure itself.

```
struct sock {
```

The first thing in the sock structure must be sock_common, described previously. Sock_common is first because tcp_w_bucket structure and perhaps other structures also have sock_common as the first part. This is to allow the same list processing and queuing functions to be used on both kinds of structures.

```
    struct sock_common  __sk_common;
```

Here we have some defines to make it easier to find the fields in the common part.

```
#define sk_family       __sk_common.skc_family
#define sk_state        __sk_common.skc_state
#define sk_reuse        __sk_common.skc_reuse
#define sk_bound_dev_if __sk_common.skc_bound_dev_if
#define sk_node         __sk_common.skc_node
#define sk_bind_node    __sk_common.skc_bind_node
#define sk_refcnt       __sk_common.skc_refcnt
This field is not used for TCP/IP.
```

This field is not used for TCP/IP.

```
    volatile unsigned char    sk_zapped;
```

The next field is used with the RCV_SHUTDOWN and SEND_SHUTDOWN socket options. As we will see in later chapters, when the SEND_SHUTDOWN option is set, in TCP, an RST will be sent when closing the socket.

```
    unsigned char       sk_shutdown;
```

The next field, when set, indicates that this socket has a valid write queue. This field is used with the write_space callback function. It indicates whether to call sk_write_space in the function sock-wfree. Sk_userlocks holds the SO_SNDBUF and SO_RCVBUF socket option settings.

```
    unsigned char       sk_use_write_queue;
    unsigned char       sk_userlocks;
```

Sk_lock is the individual socket lock used for socket synchronization. The next field, sk_rcvbuf, is the size of the receive buffer in bytes and is set from the SO_RCVBUF socket option.

```
    socket_lock_t       sk_lock;
    int                 sk_rcvbuf;
```

Sk_sleep is the sock wait queue, and sk_dst_cache is the pointer to the destination cache entry.

```
    wait_queue_head_t   *sk_sleep;
    struct dst_entry    *sk_dst_cache;
    rwlock_t            sk_dst_lock;
```

This field is used with the Security Policy Database (SPD). See Chapter 6 for more information.

```
    struct xfrm_policy *sk_policy[2];
```

The next four fields are for queuing. Sk_rmem_alloc is the number of committed bytes in the receive packet queue, and sk_receive_queue is the receive queue. Sk_wmem_alloc is the transmit queue committed length, and sk_write_queue is the transmit queue.

```
    atomic_t            sk_rmem_alloc;
```

```
struct sk_buff_head sk_receive_queue;
atomic_t            sk_wmem_alloc;
struct sk_buff_head sk_write_queue;
```

The next field is the number of optional committed bytes. It is used by socket filters, and the field sk_wmem_queued is the persistent write queue size.

```
atomic_t            sk_omem_alloc;
int                 sk_wmem_queued;
```

Sk_forward_alloc is the number of bytes in pre-allocated pages, and sk_ alloca tion is the allocation mode.

```
int                 sk_forward_alloc;
unsigned int        sk_allocation;
```

Sk_sndbuf is the size of the send buffer and is set with the SO_SNDBUF socket option.

```
int                 sk_sndbuf;
```

Sk_flags contains the values of the socket options SO_BROADCAST, SO_KEEPALIVE, and SO_OOBINLINE. Next, sk_no_check contains the value of the SO_NO_CHECK socket option and indicates whether to disable checksums. Sk_debug holds the SO_DEBUG socket option and sk_rcvtstamp holds the SO_TIMESTAMP socket option.

```
unsigned long       sk_flags;
char                sk_no_check;
unsigned char       sk_debug;
unsigned char       sk_rcvtstamp;
```

The value in the next field indicates whether to send large TCP segments. Sk_route_caps is the route capabilities, NETIF_F_TSO.

```
unsigned char       sk_no_largesend;
int                 sk_route_caps;
```

The value of sk_lingertime is set to TRUE if the SO_LINGER socket option is set. Sk_hashent contains a hash entry for several tables. The next field, sk_pair, is for the socket pair call and is not used for TCP/IP.

```
unsigned long       sk_lingertime;
int                 sk_hashent;
struct sock         *sk_pair;
```

The next field, sk_backlog, is the socket backlog queue. It is always used with the individual socket lock held. It requires low latency access.

```
struct {
    struct sk_buff  *head;
    struct sk_buff  *tail;
}  sk_backlog;
```

This lock is for the six callback functions at the bottom of the sock structure.

```
    rwlock_t              sk_callback_lock;
    struct sk_buff_head   dsk__error_queue;
```

Sk_prot is a pointer to the protocol handler within AF_INET family or another protocol family. Sk_err is the last error on this socket, and sk_err_soft are for errors that don't cause complete socket failure, only a socket that has timed out.

```
    struct proto        *sk_prot;
    int                 sk_err,
                        sk_err_soft;
```

The next two fields are for the current listen backlog and the maximum backlog set in the listen call. The field sk_priority holds the value of the SO_PRIORITY socket option.

```
    unsigned short      sk_ack_backlog;
    unsigned short      sk_max_ack_backlog;
    __u32               sk_priority;
```

The next field, sk_type, holds the socket type SOCK_STREAM or SOCK_DGRAM. Sk_localroute says to route locally only. It is the value of the SO_DONTROUTE socket option.

```
unsigned shortsk_type;
    unsigned char       sk_localroute;
```

The following field, sk_protocol, holds the protocol number for this socket within the AF_INET family. It is the same as the 1-byte protocol field in the IP header.

```
    unsigned char       sk_protocol;
```

The sk_peercred field is not used for TCP/IP. It is for passing file descriptors using Berkeley style credentials. The following three fields hold the value of the SO_RCVLOWAT, SO_RCVTIMEO, and the SO_SNDTIMEO socket options.

```
    struct ucred        sk_peercred;
    int                 sk_rcvlowat;
    long                sk_rcvtimeo;
    long                sk_sndtimeo;
Sk_filter is for socket filtering.
    struct sk_filter    *sk_filter;
```

The next field points to private areas for various protocols.

```
    void                *sk_protinfo;
```

Sk_slab points to the slab cache from which this sock structure was allocated, and sk_timer is the sock cleanup timer.

```
    kmem_cache_t        *sk_slab;
    struct timer_list   sk_timer;
```

Sk_stamp is the timestamp of the most recent received packet. Sk_socket points to the socket structure for this socket.

```
    struct timeval      sk_stamp;
    struct socket       *sk_socket;
    void                *sk_user_data;
```

The next field, sk_owner, points to the module that owns this socket.

```
    struct module       *sk_owner;
```

The following five fields point to callback functions for this socket. The first field, sk_state_change, is called when the state of this sock is changed, and the next field, sk_data_ready, indicates that there is data available to be processed. Sk_write_space is called when there is buffer space available for sending. Sk_error_report is called when there are errors to report and is used with MSG_ERRQUEUE. Sk_backlog_rcv is called to process the socket backlog.

```
    void                (*sk_state_change)(struct sock *sk);
    void                (*sk_data_ready)(struct sock *sk, int bytes);
    void                (*sk_write_space)(struct sock *sk);
    void                (*sk_error_report)(struct sock *sk);
    int                 (*sk_backlog_rcv)(struct sock *sk,
                         struct sk_buff *skb);
```

Finally, the last field is the destructor function for this sock instance. It is called when all the references are gone and the refcnt becomes zero.

```
    void                (*sk_destruct)(struct sock *sk);
} ;
```

When a sock structure instance is allocated from the slab, following the sock structure is the inet_sock, which contains a protocol information part for IPv6 and IPv4.

```
struct inet_sock {
    struct sock         sk;
#if defined(CONFIG_IPV6) || defined(CONFIG_IPV6_MODULE)
    struct ipv6_pinfo   *pinet6;
#endif
```

The inet_opt structure for IPv4 is discussed in Section 5.11.1.

```
    struct inet_opt   inet;
} ;
```

5.3.2 The Socket Structure

The socket structure is the general structure that holds control and states information for the socket layer. It supports the BSD type socket interface.

```
struct socket {
```

The first field contains the state of the socket and one of the socket state values shown later in Table 5.8. The socket flags are in the next field and hold the socket wait buffer state containing values such as SOCK_ASYNC_NOSPACE.

```
    socket_state        state;
    unsigned long       flags;
```

Ops points to the protocol-specific operations for the socket. This data structure is shown later in this chapter.

```
    struct proto_ops    *ops;
```

The next field, fasync_list, points to the wake-up list for asynchronous file calls. For more information, see fsync(2). File points to the file structure for this socket. We need to keep a pointer here to facilitate garbage collection.

```
    struct fasync_struct *fasync_list;
    struct file         *file;
```

Sk points to the sock structure for this socket. Wait is the socket wait queue.

```
    struct sock         *sk;
    wait_queue_head_t   wait;
```

Type is the socket type, and generally is SOCK_STREAM, SOCK_DGRAM, or SOCK_RAW. Passcred is not used for TCP/IP. It is for BSD style credential passing and holds the value of the SO_PASSCRED socket option

```
    short               type;
    unsigned char       passcred;
} ;
```

5.3.3 The Proto_ops Structure

This structure contains the family type for this particular set of socket operations. For IPv4, it will be set to AF_INET.

```
struct proto_ops {
```

Family is the address family. It is set to AF_INET for IPv4. Owner is the module that owns this socket.

```
intfamily;
struct module*owner;
```

Each of the following fields corresponds to a socket call. They are all pointers to the function implementing the protocol-specific operation.

```
    int           (*release)   (struct socket *sock);
    int           (*bind)     (struct socket *sock,
```

```
                                struct sockaddr *myaddr,
                                int sockaddr_len);
    int           (*connect)    (struct socket *sock,
                                 struct sockaddr *vaddr,
                                 int sockaddr_len, int flags);
    int           (*socketpair)(struct socket *sock1,
                                 struct socket *sock2);
    int           (*accept)     (struct socket *sock,
                                 struct socket *newsock, int flags);
    int           (*getname)    (struct socket *sock,
                                 struct sockaddr *addr,
                                   int *sockaddr_len, int peer);
    unsigned int  (*poll)       (struct file *file, struct socket *sock,
                                 struct poll_table_struct *wait);
    int           (*ioctl)      (struct socket *sock, unsigned int cmd,
                                 unsigned long arg);
    int           (*listen)     (struct socket *sock, int len);
    int           (*shutdown)   (struct socket *sock, int flags);
    int           (*setsockopt)(struct socket *sock, int level,
                                 int optname, char __user *optval,
                                 int optlen);
    int           (*getsockopt)(struct socket *sock, int level,
                                 int optname, char __user *optval,
                                 int __user *optlen);
    int           (*sendmsg)    (struct kiocb *iocb, struct socket
                                 *sock,
                                 struct msghdr *m, int total_len);
    int           (*recvmsg)    (struct kiocb *iocb, struct socket
                                 *sock,
                                 struct msghdr *m, int total_len,
                                 int flags);
    int           (*mmap)       (struct file *file, struct socket *sock,
                                 struct vm_area_struct * vma);
    ssize_t       (*sendpage)   (struct socket *sock, struct page *page,
                                 int offset, size_t size, int flags);
} ;
```

5.4 Socket Layer Initialization

The Linux networking infrastructure can support multiple protocol stacks or address families. Each supported protocol suite has an address family that is registered with the socket layer. This is how the common socket API can be used with the member protocols in diverse protocol suites. Each address family is listed in Table 5.1. In this book, we are primarily interested in TCP/IP and its address family AF_INET. AF_INET is registered during kernel initialization, and the internal hooks that connect the AF_INET family with the TCP/IP protocol suite are done during socket initialization. However, it is possible to register new protocol families dynamically with the socket layer. Chapter 6 includes more information about the general Linux kernel initialization mechanism. However, in this section, we are mainly concerned about the initialization of sockets for the AF_INET address family. We also cover how protocol registration is done for each of the member protocols in TCP/IP.

The socket layer, like any other Linux kernel facility, has an initialization function called during kernel initialization in file *linux/net/socket.c*. Note that initialization functions can be quickly found because they are all typed __init. Sock_init is called before Internet protocol registration because basic socket initialization must be done before each of the TCP/IP member protocols can register with the socket layer.

```
void __init sock_init(void)
{
        int i;
```

The first thing sock_init does is initialize the net_families array, which is the basic mechanism of socket protocol registration. Section 5.10 describes in detail how the socket layer maps each of the user socket calls to the underlying protocol.

```
for (i = 0; i < NPROTO; i++)
        net_families[i] = NULL;
```

Next, sk_init is called to initialize the slab cache for the sock data structure. This data structure, discussed earlier, contains all of the internal socket state information. Next, sock_init calls skb_init to set up the slab cache for socket buffers, or sk_buffs. Chapter 7 includes more detailed information about Linux slab memory allocation and socket buffers.

```
    sk_init();

#ifdef SLAB_SKB
    skb_init();
#endif
#ifdef CONFIG_WAN_ROUTER
    wanrouter_init();
#endif
```

Now, we build the pseudo-file system for sockets, and the first step is to set up the socket inode cache. Linux, like other Unix operating systems, uses the inode as the basic unit for filesystem implementation.

```
    init_inodecache();
```

Next, a pseudo filesystem is created for sockets called sock_fs_type by calling register_filesystem. Linux, like many operating systems, has a unified IO system, so that IO calls are transparent whether they are accessing devices, files, or sockets. We must register with the filesystem to use the IO system calls to read and write data through open sockets. Section 5.12 discusses the IO system call mapping in more detail. Now that the socket file system is built, we can mount it in the kernel.

```
register_filesystem(&sock_fs_type);
    sock_mnt = kern_mount(&sock_fs_type);
```

The remaining part of protocol initialization is done later when the function do_initcalls in main.c is executed. The last thing that sock_init does is initialize netfilters if they have been configured into the kernel.

```
#ifdef CONFIG_NETFILTER
    netfilter_init();
#endif
} .
```

5.5 Family Values and the Protocol Switch Table

As discussed earlier, the socket layer is used to interface with multiple protocol families and multiple protocols within a protocol family. After incoming packets are processed by the protocol stack, they eventually are passed up to the socket layer to be handed off to an application layer program. The socket layer must determine which socket should receive the packet, even though there may be multiple sockets open over different protocols. This is called *socket de-multiplexing*, and the protocol switch table is the core mechanism. This mechanism functions very much like the BSD operating system where the term *protocol switch table* is also often used.

Much TCP/IP member protocol initialization is discussed in Chapter 6 for IPv4 and Chapter 11 for IPv6. However, socket layer registration is discussed here because it is closely related to socket layer de-multiplexing. Linux makes a distinction between permanent and nonpermanent protocols. For example, permanent protocols include protocols such as UDP and TCP, which are a fundamental part of any functioning TCP/IP implementation. Removal of permanent protocols is not allowed; therefore, UDP and TCP canbe unregistered. However, other protocols can be added to the protocol switch table dynamically, and these protocols are considered nonpermanent. Figure 5.1 illustrates the registration process. It shows how the inet_protosw structure is initialized with proto and proto_ops structures for TCP/IP, the AF_INET family.

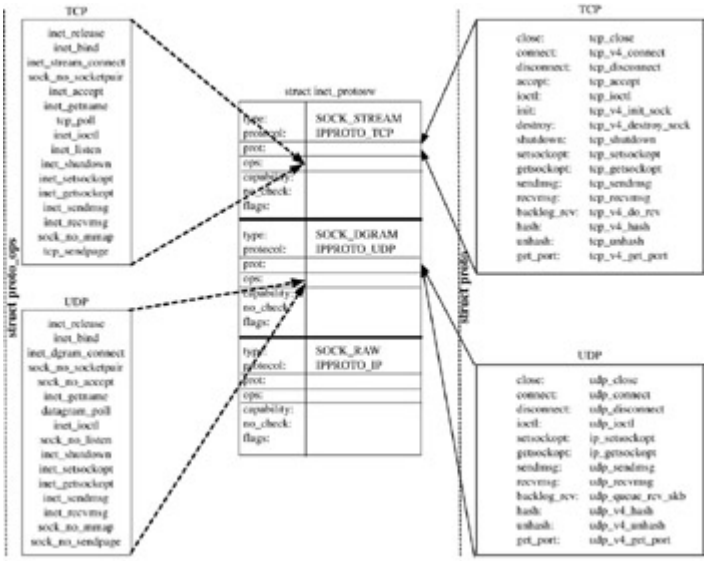

Figure 5.1: AF_INET protocol family and socket calls.

The protocol switch registration mechanism consists of two functions and a data structure for maintaining the registered protocols. One of the functions is for registering a protocol, and the other function is for unregistration. Each of the registered protocols is kept in a table called the *protocol switch table*. Each entry in the table is an instance of the inet_protosw.

The registration function, inet_register_protosw, puts the protocol described by the argument p into the protocol switch table.

```
void inet_register_protosw(struct inet_protosw *p);
```

The unregistration function, inet_unregister_protowsw, removes a protocol described by the argument p from the protocol switch table.

```
void inet_unregister_protosw(struct inet_protosw *p);
```

Each protocol instance in the protocol switch table is an instance of the inet_protosw structure, defined in file *linux/include/protocol.h*.

```
struct inet_protosw {
```

The first two fields in the structure, list and type, form the key to look up the protocol in the protocol switch table. Type is equivalent to the type argument of the socket call, and the values for this field are shown in Table 5.2.

```
    struct list_head    list;
    unsigned short      type;
```

Table 5.2: Values for Socket Types

Name	Value	Purpose
SOCK_STREAM	1	Streaming sockets. For connection-oriented or TCP sockets.
SOCK_DGRAM	2	Datagram sockets. For connectionless or UDP sockets.
SOCK_RAW	3	Raw socket. Provides application layer with direct access to network layer protocols.
SOCK_RDM	4	Reliable Delivery Message (RDM) socket.
SOCK_SEQPACKET	5	Sequential packet socket.
SOCK_PACKET	10	Sockets for direct packet access at the network device level.

The next field, protocol, corresponds to the well-known protocol argument of the socket call. This field is set to the protocol number for TCP, UDP, another protocol number, or zero for raw. This is the protocol number for the protocol that is being registered. For more information, refer to the socket call in the socket API discussion later in this chapter.

```
    int                 protocol;
```

The field prot points to the protocol block structure. This structure is used when a socket is created. This structure is used to build an interface to any protocol that supports a socket interface. The next field, ops, points to a protocol-specific set of operation functions for this protocol. The proto_ops structure is discussed in Section 5.3.3. It has the same definition as the ops field in the socket structure as discussed in Section 5.3.2.

```
    struct proto        *prot;
    struct proto_ops    *ops;
```

Capability is used to determine if the application layer program has permission for a socket operation. The appropriate level of permission is required for some socket operations with raw sockets. This is to prevent security attacks because raw socket operations can get access to internal operations of TCP/IP. See the section on Linux capabilities later in this chapter for more information.

```
    int                 capability;
```

The next field, no_check, tells the network interface driver not to perform a checksum.

```
    char                no_check;
```

Finally, the last field, flags, is defined as one of two values.

```
#define INET_PROTOSW_REUSE 0x01
#define INET_PROTOSW_PERMANENT 0x02
```

If flags is set to INET_PROTOSW_PERMANENT, the protocol is permanent and can't be unregistered. In this case, the unregistration function prints an error message and returns. For example, UDP and TCP have flags set to INET_PROTOSW_PERMANENT, and for raw sockets, (SOCK_RAW) flags value is set to INET_PROTOSW_REUSE.

```
    unsigned char       flags;
} ;
```

5.5.1 Member Protocol Registration and Initialization in IPv4

Chapter 6 contains a complete discussion of the initialization steps for IPv4. Here we will only look at socket layer registration. In the previous section, we made a distinction between permanent protocols and other protocols. Now we will look at how the permanent calls are registered with the protocol switch table. The permanent protocols in IPv4 are registered by the function inet_init, defined in file *net/ipv4/af_inet.c*.

```
static int __init inet_init(void)
{
. . .
```

The code actually registers the protocols after they have been placed into an array.

```
    for (r = &inetsw[0]; r < &inetsw[SOCK_MAX]; ++r)
        INIT_LIST_HEAD(r);

    for (q = inetsw_array; q < &inetsw_array[INETSW_ARRAY_LEN]; ++q)
        inet_register_protosw(q);
. . .
}
```

In the preceding code snippet, we can see that inet_init calls inet_register_protosw for each of the protocols in the array, inetsw_array. The protocols in the array are UDP, TCP, and raw. The values for each protocol is initialized into the inet_protosw structure at compile time as shown here.

```
static struct inet_protosw inetsw_array[] =
{
        {
```

The first protocol is TCP, so type is SOCK_STREAM and flags is set to permanent.

```
        {
                type:           SOCK_STREAM,
                protocol:       IPPROTO_TCP,
                prot:           &tcp_prot,
                ops:            &inet_stream_ops,
                capability:     -1,
                no_check:       0,
                flags:          INET_PROTOSW_PERMANENT,
        } ,
        {
```

The second protocol is UDP, so type is SOCK_DGRAM and flags is also set to permanent.

```
          {
              type:           SOCK_DGRAM,
                protocol:       IPPROTO_UDP,
                prot:           &udp_prot,
                ops:            &inet_dgram_ops,
                capability:     -1,
                no_check:       UDP_CSUM_DEFAULT,
                flags:          INET_PROTOSW_PERMANENT,
               } ,
       {
```

The third protocol is “raw,” so type is SOCK_RAW and flags is also set to reuse. Notice the protocol value is IPPROTO_IP, which is zero, and indicates the "wild card," which means that a raw socket can actually be used to set options in any protocol in the IF_INET family. This corresponds to the fact that the protocol field is typically set to zero for a raw socket.

```
              {

        type:                   SOCK_RAW,
               protocol:       IPPROTO_IP,/* wild card */
               prot:           &raw_prot,
               ops:            &inet_dgram_ops,
               capability:     CAP_NET_RAW,
               no_check:       UDP_CSUM_DEFAULT,
               flags:              INET_PROTOSW_REUSE,
                }
} ;
```

Once this registration is complete, other protocols usually implemented as modules may still add themselves to the protocol switch table at any time by calling inet_register_protocols.

5.5.2 Registration of Protocols with the Socket Layer

Chapter 6 discusses the various mechanisms for protocol registration in Linux TCP/IP, including the sequence of steps performed for Internet protocol initialization. However, this section is about important family values; that is, the registration of protocol families such as the Internet Protocol (IP) family. In this section, we cover the registration of an entire protocol family, which includes an entire suite of protocols, and how it is associated with one of the protocol families listed in Table 5.1.

For TCP/IP, this family is AF_INET. The protocol family registration step is necessary so the application programmer can access all the protocols that are part of the TCP/IP protocol family through the socket API functions. The socket layer family registration facility provides two functions and one key data structure.

The first function, sock_register, registers the protocol family with the socket layer.

```
int sock_register(struct net_proto_family *fam);
```

The family is passed in as a pointer to the net_proto_family structure, which is shown later in this section. Sock_register checks the family field in the net_proto_family structure pointed to by fam to make sure it is one of the family types listed in Table 5.1. If family is within range, it copies the argument fam into a location in the global array net_families indexed by the family value.

```
static struct net_proto_family *net_families[NPROTO];
```

Later, in this chapter, we will see how the net_families array is used to look up the protocol family associated with the domain for the requested socket.

The second function, sock_unregister, is the protocol family unregistration function.

```
int sock_unregister(int family);
```

In this function, family is the value of the protocol family, such as AF_INET. Sock_unregister reverses the registration process by setting the element in the array indexed by the parameter family to NULL. It only makes sense to use this function in a module that implements a nonpermanent protocol family. Protocol families can be written as modules so Linux provides an unregistration function. However, since IPv4 is generally statically compiled in to the kernel, sock_unregister will never be called for IPv4, the AF_INET family. IPv4 is almost always used with the Linux kernel, and it would be unusual to configure a Linux kernel without it. When protocol families are implemented as modules, the unregistration should be done from the exit function (typed __exit). See Chapter 6 for more about modules in Linux.

The data structure net_proto_family is passed as a parameter to the sock_register function.

```
struct net_proto_family
{
```

The first field, family, corresponds to one of the protocol families listed in Table 5.1, such as AF_INET.

```
    int     family;
```

The create field is a function pointer to the specific socket creation function for the protocol family specified by family.

```
    int     (*create)(struct socket *sock, int protocol);
```

It also contains a few counter fields for security support that aren't widely used.

```
    short   tauthentication;
    short   encryption;
    short   encrypt_net;
```

Finally, owner points to the module that owns this protocol family.

```
struct sockaddr_in
{
```

5.6 The Socket Application Programming Interface

The socket Application Programming Interface (API) functions are described in this section. Sockets are the fundamental basis of client server programming. Generally, socket programming follows the client-server model. At the risk of over-simplifying, we will define the server as the machine that accepts connections. In contrast, a client is the machine that initiates connections. This book won't pretend to duplicate the work of other authors on network application programming; instead, refer to [STEV98]. However, in the interest of a complete description of how the socket interface functions in Linux, the socket API functions are provided with a description of the purpose of each call. Later in this chapter, in Section 5.10, you will see what happens under the covers in the Linux TCP/IP stack when each of the functions described in this section is invoked.

Before we proceed with listing the API functions, we will discuss how IP and other addresses are passed through sockets. In general, when using the socket API, network addresses are stored in a sockaddr structure defined in file *linux/include/linux/in.h*. This structure holds different forms of address information.

```
struct sockaddr_in
{
```

Sin_port is the port number for the socket.

```
    in_port_t sin_port;
```

Sin_addr is the IP address.

```
    struct in_addr sin_addr;
} ;
```

We use the sockaddr_in structure with the socket API because sockets are generic and intended to work with a variety of protocol families and a variety of address formats, not just IPv4 with its well-known but limited 32-bit address format. This is why the sockaddr structure can vary in length depending on the format of the address it contains.

It is important to note that the terms *port* and *socket* are not synonymous. The port, along with the IP address, identifies the destination address for a packet. However, the socket is the identifier that the application uses to access the connection to the peer machine. Linux, like most Unix operating systems, provides a complete set of functions for Internet address and port manipulation.

Like most Unix operating systems before it, Linux provides a variety of IP address conversion functions for manipulation of Internet addresses. Included are functions to convert addresses between character strings and binary numbers. An example is inet_ntoa, which converts a binary IP address to a string. The Linux functions are compatible with the traditional BSD functions for network programming that have been used for many years. These conversion functions are just about indispensable for doing any type of network application programming and are too numerous to list in this section. However, you can consult the Linux-man page inet(3) for a detailed list.

The socket API deals with addresses and ports, and it is important to note that TCP/IP, like other network protocols, always considers ports and Internet addresses passed through the socket API to be in network byte order. Network byte order is the same as big endian or Motorola byte order. Linux, like other Unix-compatible OSs, provides a set of conversion functions to convert integers of various lengths between host byte order and network byte order. For a list of these functions, see the Linux man page byteorder(3).

One more thing should be mentioned before exploring the socket API functions. There are two types of sockets, one for individual datagrams and another for a streaming sequence of bytes. These two types of sockets reflect the difference between connectionless and connection-oriented service. The UDP protocol provides the connectionless or datagram service and is accessed through sockets of type SOCK_DGRAM. The TCP protocol is accessed through sockets of type SOCK_STREAM, which provide connection-oriented service. Most of the socket calls can be used with either socket type. However, one of the socket calls, connect, has slightly different semantics depending on the socket type, but connect is more commonly used with TCP. Two of the socket calls, accept and listen, are not used for UDP at all. In addition, recv, recvfrom, recvmsg, send, sendto, and sendmsg are usually used only with UDP. Generally, TCP servers and clients use write and read to move data to and from the sockets.

Now we will look at the socket API functions, the first and most important of which is socket which opens up a new connection.

```
int socket(int domain, int type, int protocol);
```

Socket must be called first before the application can use any of the networking functions of the operating system for any purpose. Socket returns an identifier also known as a socket. This identifier is essentially a file descriptor and can be used in the same way as the file descriptor returned by the open system call. In other words, read and write calls can be done by specifying the socket.

The first argument to the socket call, domain, specifies which protocol family will be accessed through the socket returned by this call. It should be set to one of the protocol families used in Linux (shown in Table 5.1) and for us it is generally AF_INET. In Linux generally, and everywhere in this book, we use the terms *protocol family* and *address family* interchangeably. The BSD derived socket implementations make a distinction between these two, but Linux does not. For compatibility with BSD, the protocol family is defined with names preceded by PF_, and the address families have names that begin with AF_, but the numerical values cor responding to each are identical. Type is generally set to one of three values for Linux TCP/IP. It is set to SOCK_STREAM if the caller wants reliable connection-oriented service generally provided by the TCP transport protocol. Type is specified as SOCK_DGRAM for connectionless service via the UDP transport protocol, or SOCK_RAW for direct network access to underlying protocols below the transport layer. The access to lower layer protocols is generally referred to as "raw network protocol" access. The allowed values for type are shown in Table 5.2.

Finally, the protocol argument to the socket call is typically set to zero when sockets are open for conventional UDP or TCP packet transmission. In some cases, though, the protocol field is used internally by the socket layer code to determine which protocol the socket accesses if the type field is insufficient. For example, to get raw protocol access to ICMP, protocol would be set to IPPROTO_ICMP and type is set to SOCK_RAW.

The next socket API call, bind, is called by applications that want to register a local address with the socket. The local address generally consists of the port number and is referred to as the name of the socket. Applications that are sending UDP packets or datagrams don’t have to call bind. If they want the peer to know where the packets came from, they should call bind.

```
int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen);
```

Bind is usually used by application servers to associate an endpoint or port and address combination with a socket. Applications will call bind if they want the socket layer to know the port on which they will be receiving data. The port number and the local IP address are specified in myaddr in the form of the sockaddr structure.

The listen API function is called by a SOCK_STREAM or TCP server to let the socket layer know that it is ready to receive connection requests on the socket, s.

```
int listen (int s, int backlog );
```

Backlog specifies the length of the queue of pending connection requests while the server is waiting for the accept call to complete.

Accept is called by the application when it is ready to accept a connection request. It returns a new socket for the accepted connection. The address of the peer requesting the connection is

placed in the sockaddr structure pointed to by addr. Addrlen should point to a variable containing the size of struct sockaddr before calling accept. After the function call returns, addrlen points to the length of the new address in addr.

```
int accept (int s, struct sockaddr *addr, socklen_t *addrlen);
```

The socket call, connect, is primarily used by a client application to establish a connection-oriented or SOCK_STREAM type connection with a server using the TCP protocol.

```
int connect (int s const struct sockaddr *serv_addr, socklen_t addrlen);
```

Serv_addr specifies the address and port of the server with which the caller wants to make a connection, and addrlen is set to the length of struct sockaddr. When used with TCP, connect actually causes TCP to initiate the connection negotiation between peers. Connect may also be used with SOCK_DGRAM, but in this case it does not actually cause a connection; instead, it only specifies the peer address information so the send call can be used through the socket, s.

The socket call, socketpair, is not used for TCP/IP so it will not be discussed here. It is used for AF_UNIX domain sockets for inter-process communication within a system. We will see later as we discuss the internal structure of the socket layer that quite a bit of the complexity of sockets is due the fact that they must support AF_UNIX based inter-process communication.

The next three socket calls—send, sendto, and sendmsg—each transmit data from socket s to a peer socket. Send is used only with a socket that has been connected, either as a server or client by having previously called connect. Sendto is used with any open SOCK_DGRAM socket because it includes the arguments to and tolen, which specify the address of the destination.

```
int send(int s, const void *msg, size_t len, int flags);
int sendto (int s, const void *msg, size_t len, int flags,
const struct sockaddr *to, socklen_t tolen);
int sendmsg (int s, const struct msghdr *msg, int flags);
```

Normally, these calls will block until there is sufficient buffer space to receive the packet of the specified length. The flags argument can contain any of the following values. All the socket flags are shown with their values in Table 5.3.

```
struct msghdr {
```

Table 5.3: Socket API Function Values for Flags

Flag	Value	Purpose
MSG_OOB	1	Request out-of-bound data.
MSG_PEEK	2	Returns data from the head of the receive queue without removing it.
MSG_DONTROUTE	4	Don’t route this message. Often used when sending ICMP packets as in the ping utility.
MSG_TRYHARD	4	Not used by TCP/IP. Synonym for MSG_DONTROUTE.

Table 5.3: Socket API Function Values for Flags

Flag	Value	Purpose
MSG_CTRUNC	8	For SOL_IP internal control messages.
MSG_PROBE	0x10	For MTU discovery.
MSG_TRUNC	0x20	Truncate message.
MSG_DONTWAIT	0x40	Set if the caller wants nonblocking IO.
MSG_EOR	0x80	End of record.
MSG_WAITALL	0x100	Wait for the full length of requested data before returning.
MSG_FIN	0x200	TCP FIN
MSG_SYN	0x800	TCP SYN
MSG_CONFIRM	0x800	Confirm the validity of the path before transmitting packet.
MSG_RST	0x1000	TCP RST
MSG_ERRQUEUE	0x2000	Requests that messages be read from the error queue.
MSG_NOSIGNAL	0x4000	Don’t generate SIGPIPE signal when a connection is determined to be broken.
MSG_MORE	0x8000	Set to indicate that sender will send more data.

Msg_name contains the "name" of the socket. This is the destination IP address and port number for this message. Msg_namelen is the length of the address pointed to by msg_name.

```
void         * msg_name;
socklen_t    msg_namelen;
```

Msg_iov is an array of buffers of data to be sent or received. It is often referred to as the "scatter gather array" but it is not used only for DMA operations. Msg_iovlen is the number of buffers in the array pointed to by msg_iov.

```
struct iovec * msg_iov;
size_t       msg_iovlen;
```

The following two fields are used for Posix 1003.1g ancillary data object information. The ancillary data consists of a sequence of pairs of cmsghdr and cmsg_data pairs. Msg_control is used to support the cmsg API function to pass control information to the underlying protocols. See the man page, cmsg(3) for more information.

```
void          * msg_control;
socklen_t     msg_controllen;
```

Msg_flags contains flags for the received message.

```
int           msg_flags;      /* flags on received message */
```

```
    } ;
```

For more information, see Chapter 4, which discusses details about rtnetlink and netlink address family and how these are used to pass control information to underlying protocols.

Struct iovec is used in the msghdr structure described earlier. It is defined in file *linux/include/linux/uio.h*.

```
struct iovec
{
```

Iov_base is a pointer to the buffer's base address. In BSD implementations, the field is typed caddr_t.

```
    void          *iov_base;
```

Size of the buffer.

```
    __kernel_size_t iov_len;
} ;
```

The next three socket calls—recv, recvfrom, recvmsg—receive a message from a peer socket. As is the case with the socket call, send, recv is generally used with a connected socket.

```
int recv (int s, void *buf, size_t len, int flags);
int recvfrom ( int s, void *buf, size_t len, int flags, struct sockaddr
*from, socklen_t *fromlen);
int recvmsg ( int s, struct msghdr *msg, int flags);
```

Normally, these calls block until data is available of the specified length. Each call returns the length of the data read from the socket. After recvfrom returns, the parameters from and fromlen contain the sender's address and length. Often these three calls are used with select to let the caller know when data is available. The argument flags may have one or more values from the list in Table 5.3. All the socket flags are shown with their values in Table 5.3, shown earlier.

The two socket calls getsockopt and setsockopt are provided so the caller can access options or settings in the underlying protocols.

```
int getsockopt ( int s, int level, int optname, void *optval, socklen_t
*optlen );
int setsockopt ( int s, int level, int optname, const void *optval,
socklen_t optlen );
```

Level should be set to one of the values from Table 5.4, each of which has the same values as the 1-byte protocol field in the IP header or the next header field of the IPv6 header. Generally, these values correspond with the 1-byte assigned numbers in the IANA database for IP protocol numbers. However, there are three exceptions: SOL_SOCKET, SOL_RAW, and SOL_IP. The value SOL_SOCKET indicates that the options settings refer to internal settings in the socket layer itself. SOL_RAW and SOL_IP indicate settings for IP internal protocols. See Section 3.11 in Chapter 3 for more about Internet number assignments.

Table 5.4: Values for Level Argument in Setsockopt System Call

Name	Value
SOL_IP	0
SOL_SOCKET	1
SOL_TCP	6
SOL_UDP	17
SOL_IPV6	41
SOL_ICMPV6	58
SOL_RAW	255

The optname argument is set to one of the values shown in Table 5.5. These values are defined in file *linux/include/asm-i386/socket.h*. Before the socket layer allows certain option values to be set, it checks to ensure that the user process has the appropriate level of permissions. These permissions are called capabilities and are described later in this chapter.

Table 5.5: Values for optname in Getsockopt and Setsockopt Calls

Name	Value	Purpose
SO_DEBUG	1	This option enables socket debugging. The option is only valid when the calling program has the CAP_NET_ADMIN capability. For more information on Linux capabilities, see the section on Linux capabilities later in this chapter.
SO_REUSEADDR	2	This option is set to allow reuse of local addresses with bind socket call.
SO_TYPE	3	This option is used with getsockopt call. When this option is used, the socket type is returned.
SO_ERROR	4	This option is also used with getsockopt call. It is for getting and clearing the current socket error.
SO_DONTROUTE	5	When this option is set, packets are not routed. They are sent only to internal destinations and hosts that are directly connected.
SO_BROADCAST	6	This option is only used with UDP. When set, it allows packets to be sent to or received from a broadcast address.
SO_SNDBUF	7	This option and the next option are used to set the maximum size for the socket send or receive buffers. See Chapter 6 for the related sysctl values.
SO_RCVBUF	8	See the SO_SNDBUF option above.
SO_KEEPALIVE	9	Generally used by TCP. Setting this option keeps socket connections alive by sending keep-alive probes. See Chapter 9 for more details.

Table 5.5: Values for optname in Getsockopt and Setsockopt Calls

Name	Value	Purpose
SO_OOBINLINE	10	This option is primarily used by TCP. It controls how out-of-band (OOB) data is handled. If the option is set, the OOB data is included in the data stream instead of being passed when the MSG_OOB flag is set.
SO_NO_CHECK	11	This flag is an undocumented flag that can be used to turn off checksums on UDP packets. It is not recommended for use by the average application program. The option is set to UDP_CSUM_DEFAULT, which enables all checksums; UDP_CSUM_NOXMIT, which disables transmit checksums; or UDP_CSUM_NORCV, which disables checksum calculation on received UDP packets.
SO_PRIORITY	12	This option is used to set the IP type-of-service (TOS) flags for outgoing packets.
SO_LINGER	13	This option is used with TCP. When set, a close(2) or shutdown(2) call on this socket will block until all queued messages have been sent and received by the peer or the number of seconds specified in the timeout has elapsed.
SO_BSDCOMPAT	14	This option is used by UDP. It specifies that ICMP errors received on this socket will not be passed to the application via this socket.
SO_PASSCRED	16	Used for passing file descriptors via control message. Not generally used by AF_INET type sockets.
SO_PEERCRED	17	This option is only used for AF_UNIX type sockets.
SO_RCVLOWAT	18	This and the next option are used with getsockopt. They obtain the size of the buffer necessary before data is passed to TCP. In Linux, they are always set to 1 byte.
SO_SNDLOWAT	19	See SO_RCVLOWAT option.
SO_RCVTIMEO	20	These two options specify the amount of time to wait before receiving or sending an error.
SO_SNDTIMEO	21	See SO_RCVTIMEO option.

5.7 Packet, Raw, Netlink, and Routing Sockets

To complete any discussion of Socket application layer programming, we must include information about Linux's special sockets. These sockets are for internal message passing and raw protocol access. Netlink, routing, packet, and raw are all types of specialized sockets. Netlink provides a socket-based interface for communication of messages and settings between the user and the internal protocols. BSD-style routing sockets are supported by Linux netlink sockets. This is why, as shown in Table 5.1, AF_ROUTE and AF_NETLINK are identical. AF_ROUTE is provided for source code portability with BSD Unix. Rtnetlink includes extensions to the messages used in the regular netlink sockets. Rtnetlink is for application-level management of the neighbor tables and IP routing tables. Section 5.9 includes more details about the internal implementation of netlink and rtnetlink sockets and how they interface to the protocols in the AF_INET family.

Packet sockets are accessed by the application when it sets AF_PACKET in the family field of the socket call.

```
ps = socket (PF_PACKET, int type, int protocol);
```

Type is set to either SOCK_RAW or SOCK_DGRAM. Protocol has the number of the protocol and is the same as the IP header protocol number or one of the valid protocol numbers.

Raw sockets allow user-level application code to receive and transmit network layer packets by intercepting them before they pass through the transport layer. This type of socket is generally not used for link or physical layer access because the link layer headers are stripped from received packets before delivering them to the socket.

```
rs = socket ( PF_INET, SOCK_RAW, int protocol);
```

Protocol is set to the protocol number that the application wants to transmit or receive. A common example of the use of raw sockets is the ping command, ping(8). Ping is an application that accesses the ICMP protocol, which is internal to IP and does not register directly with the socket layer. The ping command sends ICMP echo request packets and listens for echo replies. When the ping application code opens the socket, it sets the protocol field in the socket call to IPPROTO_ICMP. Ping and other application programs for route and network maintenance make use of a Linux utility library call to convert a protocol name into a protocol number, getprotent(3).

Netlink sockets are accessed by calling socket with family set to AF_NETLINK.

```
ns = socket (AF_NETLINK, int type, int netlink_family);
```

The type parameter can be set to either SOCK_DGRAM or SOCK_STREAM, but it doesn't really matter because the protocol accessed by is determined by netlink_family, and this parameter is set to one of the values in Table 5.6. The send and recv socket calls are generally used with netlink. The messages sent through these sockets have a particular format. See the netlink(7) for more details. There is quite a bit more complexity to using the netlink sockets than

we are describing in this section. Later in this chapter, we will discuss the internal structure of netlink in more detail while we see how the netlink mechanism interfaces to the protocols in TCP/IP.

Table 5.6: Values for the Netlink_family Argument

Name	Value	Purpose
NETLINK_ROUTE	0	Routing or device hook
NETLINK_SKIP	1	Reserved for ENskip
NETLINK_USERSOCK	2	Reserved for future user-mode socket protocols
NETLINK_FIREWALL	3	Hook for access to firewalling hook
NETLINK_TCPDIAG	4	For TCP socket monitoring
NETLINK_NFLOG	5	Netfilter and iptables user logging
NETLINK_ARPD	8	For access to the ARP table
NETLINK_ROUTE6	11	Af_inet6 route communications channel
NETLINK_IP6_FW	13	Access to packets that fail IPv6 firewall checking
NETLINK_DNRTMSG	14	DECnet routing messages
NETLINK_TAPBASE	16	Values from 16 to 31 specify ethertap device instances

Routing sockets are specified by the AF_ROUTE address family. In Linux, routing sockets are identical to netlink sockets.

Rtnetlink extends netlink sockets by appending netlink type messages with some additional attributes. Rtnetlink sockets are most often used for application layer access to the routing tables.

These special socket types can be found in the man pages. Refer to the man pages netlink(7), rtnetlink(7), packet(7), and raw(7) for more information.

5.8 Security and Linux Capabilities

If there were no security mechanism, the special sockets discussed in the previous section could be used by programmers to gain almost complete access to underlying kernel structures, including the internals of the TCP/IP stack. In the wrong hands, this rich set of functions could be a vehicle for security violations. We want to prevent unauthorized users from engaging in Denial-of-Service (DoS) attacks by deleting routes or rerouting sockets. Linux capabilities are the mechanism used in recent versions of the kernel for defining levels of access. Traditional Unix systems had two levels of access, either root or user. With root access, you could do anything. More recently, however, Linux has implemented the POSIX.1e Draft Capabilities that divide the complete set of root-level privileges into subsets. The Linux capabilities is the mechanism for holding and granting the permissions. We won't go into a detailed discussion of all the POSIX capabilities because it isn't relevant to TCP/IP. Refer to the man page cap_init(3) for more details. We will cover how they are used to control access to sockets.

Capabilities are important because it stores the user-level permissions that are checked by the socket layer before allowing raw or netlink socket access. To perform these checks, Linux provides an internal function, capable, defined in file *linux/include/linux/sched.h* for checking capabilities against the currently executing user-level process.

```
extern int capable(int cap);
```

Starting with version 2.4, Linux provides a set of data structure and macros in the file *linux/include/apability.h* to hold and manipulate the capabilities. If security is configured into the kernel, CONFIG_SECURITY, the capable function points to an operation that is part of a plug-in to the Linux 2.6 security framework. If not, the function capable is an inline that can be called from either inside the kernel or a user-level process. It returns the effective capabilities of the current process, the one that made the user-level socket call. This is how capable is implemented as an inline.

```
static inline int capable(int cap)
{
    if (cap_raised(current->cap_effective, cap)) {
        current->flags |= PF_SUPERPRIV;
        return 1;
    }
    return 0;
}
```

In Linux, the global variable current always points to the currently executing process. The kernel capabilities are stored in a 32-bit integer, kernel_cap_t. Current stores the following three types of capabilities.

```
kernel_cap_t   cap_effective, cap_inheritable, cap_permitted;
```

However, in the socket layer code, we only actually check the cap_effective capabilities because we are interested in the capabilities of the current application process making the socket call. See the file *linux/include/linux/capability.h* for a list of all the POSIX capabilities and the numerical values associated with each.

5.9 A Note about the Socket API and IPv6

Chapter 11 covers the Linux IPv6 protocol, how it is implemented, and how it compares to the IPv4 implementation. However, in this section, we will mention the address family used with the IPv6 and a few changes that were made to the 2.6 kernel socket API to accommodate the protocol suite. IPv6 introduces a new address family, AF_INET6, which is defined in *linux/include/socket.h* along with the other address families. In addition, there is a new socket address type for IPv6, sockaddr_in6, defined in file *linux/include/linux/in6.h*.

```
struct sockaddr_in6 {
```

The first two fields look very much like the sockaddr_in structure. The following field, sin6_family, is set to the value AF_INET6 or 10.

```
unsigned short intsin6_family;
```

This field, sin6_port, is the port number for either UDP or TCP. The next field, sin6_flowinfo, is the IPv6 flow information. See Chapter 11 for more information about this IPv6 flow.

```
    __u16               sin6_port;
    __u32               sin6_flowinfo;
```

The next field, sin6_addr, is the actual IPv6 address defined as a union. The last field is the scope ID, which defines the scope of the address. We discuss this structure in more detail and the IPv6 address scope in Chapter 11.

```
    struct in6_addr     sin6_addr;
    __u32               sin6_scope_id;
} ;
```

IPv6 sockets are backward compatible with IPv4. IPv6 specifies that data can be sent either via IPv4 or IPv6 through any open IPv6 socket. The socket API is defined so that application code that transmits data over IPv6 will also be compatible with IPv4 without modification. Underneath the covers, the actual 128-bit IPv6 address has a subtype that includes the 32-bit IPv4 address. In Chapter 11, we examine IPv6 addressing in more detail.

In addition, to aid programmers in writing code that is independent of either protocol type, Linux provides API library functions. These functions convert addresses between the IPv4 and IPv6 formats.

The first function, getaddrinfo(3), is for network address and service translation.

```
int getaddrinfo (const char *node, const char *service, const struct
addrinfo *hints, struct addrinfo **res);
```

It is a generic function that combines the functionality of three other functions: getipnodebyaddr(3), getservbyname(3), and getipnodebyname(3). Getaddrinfo creates either IPv4 or IPv6 type address structures for use with the bind or connect socket calls. If not NULL, hints specifies the preferred socket type. It points to an instance of addrinfo, the fields of which determine the socket type. Either or both of the next two parameters, node or service, may be specified, but only one of them can be NULL. Node specifies an address in IPv4 format, an address in IPv6 format, or a hostname. Service specifies the port number. Getaddrinfo supports multiple addresses and multihoming; therefore, the result res is a linked list of addrinfo structures.

The data structure, addrinfo, is used in both the hints and result for the getaddrinfo(3) library function. If used for hints, addrinfo specifies the preferred address family AF_INET, AF_INET6, or AF_UNSPECIFIED.

```
struct addrinfo {
    int           ai_flags;
    int           ai_family;
    int           ai_socktype;
    int           ai_protocol;
    size_t        ai_addrlen;
```

```
    struct sockaddr  *ai_addr;
    char         *ai_canonname;
    struct addrinfo *ai_next;
}
```

The next function, freeaddrinfo(3) , deletes the linked list of addrinfo structures, pointed to by res, which were created by getaddrinfo (3).

```
void freeaddrinfo (struct addrinfo *res );
```

The last function that we will discuss is getnameinfo(3) .

```
int getnameinfo (const struct sockaddr *sa, socklen_t salen, char
*host, size_t hostlen, char *serv, size_t servlen, int flags);
```

The parameter sa points to a generic socket address structure and can be either sockaddr_in or sockaddr_in6. Getnameinfo is really a generalized function for address to node name translation that can work with the address formats in both IPv4 and IPv6. It converts numerical address to and from text host names in a way that is independent of IPv4 and IPv6.

5.10 Implementation of the Socket API System Calls

Many operating systems designed for embedded systems are implemented in a flat memory space. These operating systems were originally designed for CPUs that don't have a Memory Management Unit (MMU) that maps physical memory to virtual memory. In these systems, which are generally smaller, the operating system kernel functions can be called directly from application-level programs without doing any memory address translation. In contrast, Linux is a virtual memory operating system. It requires a processor with an MMU. In a virtual memory operating system, each user process runs in its own virtual address space. The socket API functions are included in the system calls. These system calls are different from ordinary library calls because they can be used in nonblocking mode so that the calling user process does not have to wait while the operating system completes the processing of the request. In addition, arguments in the function call are in the memory space of the user process and must be mapped into kernel space before they can be accessed by any kernel-level code. In our case, these arguments point to data to be sent and received through the TCP/IP protocol stack.

The socket API supports other protocol families besides AF_INET or TCP/IP, and all the protocol families supported by Linux are shown in Table 5.1. In addition, Linux provides the capability of defining a new module containing an unknown protocol family. Because of this, there are several steps involved with directing each application layer socket call to the specific protocol that must respond to the request. This is a complex process and has several steps. First, any address referenced in the call's arguments must be mapped from user space to kernel space. A complete discussion of the Linux virtual memory architecture is beyond the scope of this book. However, in Chapter 7 we do discuss the Linux slab cache system of memory allocation as part of the examination of socket buffers. Next, the functions themselves must be translated from generic socket layer functions to the specific functions for the protocol family. Finally, the functions must be translated from the protocol family generic functions to the specific functions for the member protocol in the family.

As we shall see in this section, each of the socket API calls is mapped to a set of corresponding calls in the kernel with a sys_ in front of the name, and each of these calls are defined in the file *linux/net/socket.c*. Most of these functions don't do much other than call the address family-specific function through the ops field in the socket structure. Sys_socket is discussed separately in another section because it is quite complex in that it creates a new socket and sets up the structures to allow the other socket API functions work.

Earlier in this chapter, we covered each of the socket API functions, but in this section, we will see what happens under the hood when the socket API functions are called. Figure 5.1 shows how the application layer socket calls are mapped to the corresponding protocol-specific kernel functions. When any of the socket API functions are called, it causes an interrupt to the kernel's syscall facility, which in turn calls the sys_socketcall function in the file *linux/net/socket.c*. In many CPU architectures such as the Intel x86 family, pointers to the system call's arguments are passed to the kernel in one or more CPU registers. The implementation of the socket call itself is discussed in Section 5.7. The implementation of the other socket function is discussed in this section.

5.10.1 The Socket Multiplexor

The purpose of the socket multiplexor is to unravel the socket system calls. This mechanism is implemented in the file *linux/net/socket.c* and consists largely of the function sys_socketcall. This function maps the addresses in the arguments from the user-level socket function to kernel space and calls the correct kernel call for the specified protocol.

```
asmlinkage long sys_socketcall(int call, unsigned long __user *args);
```

The first thing it does is map each address from user space to kernel space. It does this by calling copy_from_user. Next, sys_socketcall invokes the system call function that corresponds to a user-level socket call. For example, when the user calls bind, sys_socketcall maps the user-level bind to the kernel function, sys_bind, and listen is mapped to sys_listen. Each of these socket system call functions is also defined in the file *linux/net/socket.c*. Each of these functions returns a file descriptor (fd). The fd is also referred to as a socket. To support standard IO, sockets and file descriptors are treated as the same thing from the IO call's point of view. The fd serves as a handle to reference the open socket. Each socket API function includes the file descriptor for the open socket as the first parameter. When a socket API function is called, in the kernel, we use fd to fetch a pointer to the socket structure that was originally created when the socket was opened. Once we have a pointer to the socket structure, we retrieve the function specific to the address family and protocol type through the open socket. To do this, we call the protocol- specific function through a pointer in the structure pointed to by the ops field of the socket structure. The socket structure is discussed in Section 5.3.2, and the proto_ops structure in Section 5.3.3.

For an example of how the socket calls are mapped, we will look at the specific socket API call, send. When the application calls send, the kernel translates this call to sys_send, which calls sock_sendmsg, which in turn calls the sendmsg function for the protocol by dereferencing the proto_ops structure in the socket.

```
return sock->ops->sendmsg(iocb, sock, msg, size);
```

Sys_bind, sys_listen, and sys_connect do little other than call the address family's function. In addition, sys_getname and sys_getpeername both map to the address family's function for the open socket, fd.

Sys_send does nothing more than call sys_sendto. Instead of calling directly into the protocol, sys_sendto calls the socket layer function sock_sendmsg, described in Section 5.6.2. In addition, sys_recv calls sys_recvfrom directly. Sys_recvfrom calls the socket layer function, sock_recvmsg.

Sys_setsockopt and sys_getsockopt check the level argument. If level is set to SOL_SOCKET, one of the socket layer functions, sock_setsockopt or sock_getsockopt is called. If level is anything else, the respective protocol-specific function is called through the proto_ops structure accessed by the ops field of the socket structure.

Sys_shutdown also calls the corresponding protocol specific function through the shutdown field in the proto_ops structure.

Sys_sendmsg and sys_recvmsg have a bit more work to do than the other socket functions. They must verify that the iovec buffer array contains valid addresses first. Each address is mapped from kernel to user space later when the data is actually transferred but the addresses are validated now. After completing the validation of the iovec structure, sock_sendmsg and sock_recvmsg functions are called, respectively.

Sys_accept is a bit more complicated because it has to establish a new socket for the new incoming connection. The first thing it does is call sock_alloc to allocate a new socket. Next, it has to get a name for the socket by calling the function pointed to by the getname field in the ops field in the socket structure. Remember that the "name" of a socket is the address and port number associated with the socket. Next, it calls sock_map_fd to map the new socket into the pseudo socket filesystem. Refer to Section 5.7 to see how this works because it is very similar to what the socket call does.

5.10.2 Implementation of Socket Layer Internal Functions

In the previous section, we showed how the application layer calls to the socket system calls. Next, we discussed how these calls get resolved to the internal socket layer functions. Now, we will look at some of these internal socket layer functions in more detail.

The first two functions in the socket layer, sock_sendmsg and sock_recvmsg, will be discussed briefly. They are called from sys_sendmsg and sys_recvmsg, respectively. These two functions are implemented in the socket layer to support the sending and receiving of "credentials." This is a Unix-compatible method of passing file descriptors among user-level applications. Sock_sendmsg calls scm_send before calling the protocol-specific sendmsg function, and likewise, sock_recvmsg calls scm_recv after calling the protocol-specific recvmsg function. The functions scm_send and scm_recv are defined in *linux/net/core/scm.c* and implement the "credentials."

Next, we will discuss the functions sock_read, sock_write, sock_fcntl, sock_close, and sock_ioctl. These functions are also defined in file *linux/net/socket.c* and are actually socket layer implementations of the IO system calls. Each of these functions is called with a file pointer in the argument file. They first call the socki_lookup function to get the socket structure, and then call the socket layer function with a pointer to the socket structure. In Section 5.9 we saw how the file IO system calls are mapped to sockets. The functions, sock_read and sock_write set up an iovec type msghdr structure before calling sock_recvmsg and sock_sendmsg, respectively.

Sock_setsockopt and sock_getsockopt are called from the system call if level is set to SOL_SOCKET. The purpose of these functions is to set values in the sock structure according to the options that were passed as a parameter by the application layer. The system-level functions, sys_setsockopt and sys_getsockopt call these socket layer functions before any protocol-specific settings are altered.

Sock_setsockopt gets a pointer to the sock structure from the sk field of the socket structure, sock, which was passed as an argument. Next, it sets options in the sock structure, sk, based on the values pointed to by the optname and optval arguments. Refer to Section 5.3.1 for a description of the fields in the sock structure. If SO_DEBUG is set in optname, debug is set, reuse is set to the value of SO_REUSEADDR, localroute to the value of SO_DONTROUTE, no_check is set to the value of SO_NO_CHECK, and priority to the value of SO_PRIORITY. In addition, bsdism is set to the value of SO_BSDCOMPAT, passcred to the value of SO_PASSCRED, rcvtstamp to the value of SO_TIMESTAMP, and rcvlowat to the value of SO_RCVLOWAT. If SO_SNDBUF or SO_RCVBUF are set, sndbuf or rcvbuf are set to minimum values or two times optval, whichever is greater. The priority field in sk is set to optval if SO_PRIORITY is set in optname, but capabilities are checked first.

A few other fields in sk are handled specially. If SO_KEEPALIVE is set in optname, and the protocol field in sk is equal to IPPROTO_TCP, tcp_set_keepalive is called with the value in optval. Remember that the protocol field of the sock structure comes from the protocol argument in the socket function that created the socket. If the SO_LINGER option is present, the linger field is set and the value of the lingertime field is calculated from optval.

Sock_getsockopt reverses what sock_setsockopt does. It retrieves certain values from the sock structure for the option socket and returns them to the user. Refer to Section 5.3.1 on the sock structure to see which fields hold values for the socket options. A few options deserve special attention because they don't have a cor responding action in sock_setsockopt. For example, if SO_ERROR is set in optname, the current socket error is retrieved from the err field in the sock structure and the err field is cleared atomically. If the SO_TYPE option is set, the value of the type field is returned.

5.10.3 Implementation of Protocol Internal Socket Functions

Most of the behavior specific to the member protocols in the AF_INET family is described in later chapters. In this section, we will complete the discussion of how each member protocol communicates with the socket layer. As shown in Figure 5.2, the file descriptor fd is used to map each socket API call with a function specific to each protocol. The mechanism that sets up this mapping is described in detail in Section 5.10 for the socket system calls, and in Section 5.12 for

the IO system calls. In addition, as we saw in Section 5.5.2, each of the protocols registers itself with the protocol switch table. When the socket structure is initialized, as described in Section 5.4, the ops field was set to the set of protocol-specific operations from the entry in the protocol switch table.

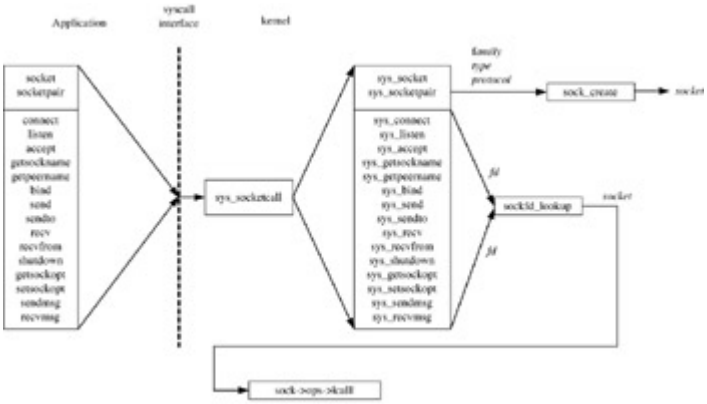

Figure 5.2: Mapping of socket calls.

Once all the complex initialization is done as described in other sections, the actual mapping is quite simple. In most cases, the “sys_" versions of the socket functions simply call sockfd_lookup to get a pointer to the socket structure and call the protocol’s function through the ops field. The function sys_getsockname in file *linux/net/socket.c* can provide us with a simple example. This function is called in the kernel when the user executes the getsockname socket API function to get the address (name) of a socket.

```
asmlinkage long sys_getsockname(int fd, struct sockaddr *usockaddr,
int *usockaddr_len)
```

Fd is the open socket. After returning, usockaddr will point to the address for the socket. usockaddr_len points to the length of the socket address.

```
{
    struct socket *sock;
    char address[MAX_SOCK_ADDR];
    int len, err;
    sock = sockfd_lookup(fd, &err);
```

This is where the protocol-specific function is called. The ops field contains the socket functions for each protocol. The protocols are UDP for SOCK_DGRAM and TCP for SOCK_STREAM.

```
    if (err)
        goto out_put;
    err = move_addr_to_user(address, len, usockaddr, usockaddr_len);
```

Here the return values are mapped into user space.

```
out_put:
    sockfd_put(sock);
```

This bumps the use count on the open “file” associated with the socket, fd.

```
out:
    return err;
}
```

5.11 Creation of a Socket

Before the user can perform any operations with the TCP/IP stack, she creates a new socket by calling the socket API function. Sockets are generic and not necessarily associated with TCP/IP, so quite a few things happen before any code in the TCP/IP stack itself gets called. Sockets can be created for many protocol types other than TCP/IP. This section will describe what happens under the hood in the Linux kernel when the function socket is called. Earlier in this chapter, we described how the socket API functions get mapped to the protocol family-specific functions. It is through this mapping that the sys_socket function executes an AF_INET specific function for the TCP/IP protocol family. First, the socket layer is responsible for activities that are not specific to a particular protocol family like AF_INET. After the generic initialization is complete, socket creation will call the socket creation function for the protocol-specific family. After we discuss the generic socket creation, we will discuss what happens during socket creation for the AF_INET protocol family.

Sock_create, defined in file *linux/net/socket.c*, is called from sys_socket. This function initiates the creation of a new socket.

```
int sock_create(int family, int type, int protocol, struct socket **res);
```

First, sock_create verifies that family is one of the allowed family types shown in Table 5.1. Then, it allocates a socket by calling sock_alloc, which returns a new socket structure, *sock*. See Section 5.3.2 for a discussion of the socket structure.

Sock_alloc, called from sock_create, returns an allocated socket structure. The socket structure is actually part of an inode structure, created when sock_alloc calls new_inode. It is necessary to have an inode for Linux IO system calls to work with sockets. Section 5.8 explores the IO system call mapping in more detail. Most of the fields in the inode structure are important only to "real" filesystems; however, a few of them are used by sockets. The fields in the inode structure that are used for sockets are listed in Table 5.7.

Table 5.7: Inode Structure Fields Used by Sockets

Field	**Prototype**	**Purpose**
u		In a socket type inode, this field contains a *socket* structure.
i_dev	kdev_t	Set to device. This is always NULL for a socket inode.
i_sock	unsigned char	Set to 1 for a socket inode.
i_mode	umode_t	Set to indicate IO permissions and interface

Table 5.7: Inode Structure Fields Used by Sockets

Field	Prototype	Purpose
		type.
i_uid	uid_t	Set to the UID of the current user process that is opening the socket.
i_gid	i_gid	Set to the GID of the current user process that is opening the socket.
i_fop	struct file_operations *	In a socket type inode, points to file operations for sockets.

Once the inode is created, sock_alloc retrieves the socket structure from the inode. Then, it initializes a few fields in the socket structure. The Inode field is set to point back to the inode structure containing this socket structure, and fasync_list is set to NULL. Fasync_list is for supporting the fsync system call for synchronizing the in-memory portions of a file with permanent storage. For a socket, fsync will flush the buffered data in the socket layer. Posix.1b defines the behavior of the fsync system call. See [GALL95] for more information about Posix.1b.

Sockets maintain a state related to whether an open socket represents a connection to a peer or not. These states are maintained in the field in the socket structure called, state, which is initialized to SS_UNCONNECTED when the socket is created. See Table 5.8 for a description of the socket states. These states are defined in the file, *linux/include/linux/net.h*.

Table 5.8: Socket States

State Name	Value	Description
SS_FREE	0	Socket is not allocated yet.
SS_UNCONNECTED	1	Unconnected to another socket.
SS_CONNECTING	2	Socket is in the process of connecting.
SS_CONNECTED	3	Connected to another socket.
SS_DISCONNECTING	4	Socket is in the process of disconnecting.

It is important to remember that sockets are not just for TCP/IP. The states maintained in the socket structure do not contain all the same states as TCP, which is quite a bit more complicated. See Chapters 8 and 10 for a completion of TCP. The socket state really only reflects whether there is an active connection. Sockets support other protocol families and must be generic. The internal logic to manage the protocol attached to a new socket will be maintained in the protocol itself, and TCP is no exception. However, since the socket layer must support several protocols, it requires some internal connection management logic in addition to what is in the internal TCP implementation.

For now, we set ops to NULL. Later, when the family member protocol-specific create function is called, ops will be set to the set of protocol-specific operations. Sock_alloc initializes a few more fields before returning. The flags field is initialized to zero because no flags are set yet.

Later, the application will specify the flags by calling one of the send or receive socket API functions. Table 5.3 contains a description of the flags used with the socket API. Finally, two other fields, sk and file, are initialized to NULL, but they both deserve a little attention. The first of these fields, sk, will be set later to point to the internal sock structure by the protocol-specific create function. File will be set to a file pointer allocated when sock_map_fd is called. The file pointer is used to maintain the state of the pseudo file associated with the open socket.

After returning from the sock_alloc call, sock_create calls the create function for the protocol family. It accesses the array net_families to get the family's create function. For TCP/IP, family will be set to AF_INET.

5.11.1 Creation of an AF_INET Socket

The create function for the TCP/IP protocol family, AF_INET is inet_create, is defined in file *af_inet.c*.

```
static int inet_create(struct socket *sock, int protocol);
```

Inet_create is defined as static because it is not called directly. Instead, it is called through the create field in the net_proto_family structure for the AF_INET protocol family. Inet_create is called from sys_socket when family is set to AF_INET. In inet_create, we create a new sock structure called sk and initialize a few more fields. The new sock structure is allocated from the slab cache, inet_sk_slab. Linux has multiple inet_sk_slabs, one for each protocol. Linux slab caches are more efficient if they are specific for the purpose, so most fields can be pre-initialized to common values. We call sk_alloc to allocate the sock structure from the slab cache that is specific to the protocol for this socket.

```
sk = sk_alloc(PF_INET, GFP_KERNEL, inet_sk_size(protocol)
inet_sk_slab(protocol));
```

Sk points to the new slab cache. Next, inet_create searches the protocol switch table to look for a match from the protocol.

After getting the result from the search of the protocol switch table, the capability flags are checked against the capabilities of the current process, and if the caller doesn't have permission to create this type of socket, the user level socket call will return the EPERM error.

Now, inet_create will set some fields in the new sock data structure, however, many fields are pre-initialized when allocation is done from the slab cache. The field sk_family is set to PF_INET. The prot field is set to the protocol's protocol block structure that defines the specific function for each of the transport protocols. No_check and ops are set according to their respective values in the protocol switch table. If the type of the socket is SOCK_RAW, the num field is set to the protocol number. As will be shown in later chapters, this field is used by IP to route packets internally depending on whether there is a raw socket open. The sk_destruct field of sk is set to inet_sock_destruct, the sock structure destructor. The sk_backlog_rcv field is set to point to the protocol-specific backlog receive function. Next, some fields in the protocol family-specific part of the sock structure are initialized. As discussed in Section 5.3.1, the sock structure is followed by a protocol-specific portion for each of the two protocol families, IPv4 and IPv6,

and this part is accessed through a macro, inet_sk, which is defined in the file *linux/include/linux/ip.h*.

```
#define inet_sk(__sk) (&((struct inet_sock *)__sk)->inet)
```

The inet_opt structure for IPv4 is also defined in the file *linux/include/linux/ip.h*.

```
struct inet_opt
{
```

These first few fields are for socket layer de-multiplexing of incoming packets. Daddr is the peer IPv4 address, and rcv_saddr is the bound local IPv4 address. Dport is the destination port, num is the local port, and saddr is the source address.

```
    __u32               daddr;
    __u32               rcv_saddr;
    __u16               dport;
    __u16               num;
    __u32               saddr;
```

The uc_ttl field is for setting the time-to-live field in the IP header.

```
    int                 uc_ttl;
```

Tos is for setting the type of service field in the IP header. The cmsg_flags field is used by setsockopt and getsockopt to communicate network layer IP socket options to the IP protocol layer. In addition, ip_options points to the values associated with the options set in the cmsg_flags field.

```
    int                 tos;
    unsigned            cmsg_flags;
    struct ip_options   *opt;
```

Sport is the source port number.

```
    __u16               sport;
```

Hdrincl is for raw sockets. It states that the IP header is included in the packet delivered to the application level.

```
    unsigned char       hdrincl;
```

This field is the multicasting time-to-live.

```
    __u8                mc_ttl;
```

The next field indicates whether multicast packets should be looped back. Pmtudisc indicates whether MTU discovery should be performed on this interface.

```
    __u8                mc_loop;
```

```
    __u8                pmtudisc;
```

The next field, id, contains the counter for identification field in the IP header. This field is used by the receiving machine for re-assembling fragmented IP packets.

```
    __u16               id;
    unsigned            recverr : 1,
                        freebind : 1;
```

Mc_index is the index for the output network interface used for transmission of multicast packets, and mc_addr is the source address used for outgoing packets sent to a multicast address.

```
    int                 mc_index;
    __u32               mc_addr;
```

This field, mc_list, points to the list of multicast address groups to which the interface has subscribed.

```
    struct ip_mc_socklist *mc_list;
```

The next field is the cached page from the sendmsg socket function, and sndmsg_off is the offset into the page.

```
    struct page         *sndmsg_page;
    u32                 sndmsg_off;
```

The following structure keeps information about the IP options needed to build an IP header on each outgoing IP fragment. Since all the fragments have almost identical headers, the options are kept here to speed the process of building IP headers on consecutive fragments. It is called cork, because the socket is "corked," waiting for all fragments of the total IP datagram to be transmitted.

```
    struct {
        unsigned int    flags;
        unsigned int    fragsize;
        struct ip_options *opt;
        struct rtable   *rt;
```

This field, length, is the total length of all frames in the fragmented IP datagram.

```
        int             length;
        u32             addr;
        struct flowi    fl;
    }  cork;
} ;
```

Some fields in the inet_opt are initialized by inet_create. Some of these fields are related to multicast transmission, which is covered in more detail in Chapter 9. In each case, inet points to the instance of the inet_opt structure shown previously.

```
    inet->uc_ttl    = -1;
```

```
inet->mc_loop  = 1;
inet->mc_ttl   = 1;
inet->mc_index = 0;
inet->mc_list  = NULL;
```

The default time-to-live field, mc_ttl is initialized because this value will be used in the time-to-live IP header field for multicast packets. Even though these values are initialized here, the application may change the values in these fields later through the setsockopt call. Finally, inet_create calls the protocol-specific initialization function through the init field in the protocol block structure, proto, defined in file *linux/include/net/sock.h*.

Here is the proto structure.

```
struct proto {
```

Most of the fields in this structure point to the protocol-specific operations. We will explain a few of these functions in a little more detail.

```
    void                (*close)(struct sock *sk,
                                 long timeout);
    int                 (*connect)(struct sock *sk,
                                   struct sockaddr *uaddr,
                                   int addr_len);
    int                 (*disconnect)(struct sock *sk, int flags);

    struct sock *       (*accept) (struct sock *sk, int flags, int *err);

    int                 (*ioctl)(struct sock *sk, int cmd,
                                 unsigned long arg);
```

Init points to the protocol's specific initialization function. This function is called when a socket is created for this protocol. Destroy points to the destructor function for this protocol. The destructor is executed when a socket for this protocol is closed.

```
    int                 (*init)(struct sock *sk);
    int                 (*destroy)(struct sock *sk);
    void                (*shutdown)(struct sock *sk, int how);
    int                 (*setsockopt)(struct sock *sk, int level,
                                      int optname, char *optval, int optlen);
    int                 (*getsockopt)(struct sock *sk, int level,
                                      int optname, char *optval,
                                      int *option);
    int                 (*sendmsg)(struct kiocb *iocb, struct sock *sk,
                                   struct msghdr *msg, int len);
    int                 (*recvmsg)(struct kiocb *iocb, struct sock *sk,
                                   struct msghdr *msg,
                                   int len, int noblock, int flags,
                                   int *addr_len);
    int                 (*sendpage)(struct sock *sk, struct page *page,
                                    int offset, size_t size, int flags);
    int                 (*bind)(struct sock *sk,
                                struct sockaddr *uaddr, int addr_len);

    int                 (*backlog_rcv) (struct sock *sk,
```

```
                                        struct sk_buff *skb);
```

The following three functions are for keeping track of sock structures, looking them up, and getting the port number associated with the sock, respectively.

```
    void                (*hash)(struct sock *sk);
    void                (*unhash)(struct sock *sk);
    int                 (*get_port)(struct sock *sk, unsigned short snum);
    char                name[32];
```

Inuse indicates whether this sock structure is being used. For SMP implementations, there is one per CPU.

```
    struct {
        int       inuse;
        u8          __pad[SMP_CACHE_BYTES - sizeof(int)];
    }  stats[NR_CPUS];
} ;
```

The sk_prot field in the sock structure points to the protocol block structure. The init field in the proto structure is specific for each protocol and socket type within the AF_INET protocol family.

5.11.2 Socket Lockets—Individual Socket Locks

Each open socket has a locking mechanism to eliminate contention problems between the kernel main thread and the "bottom half," or the various tasklets, timers, and interrupt handlers. As we know from earlier discussion, each open socket contains an instance of the sock structure. The individual socket lock is in the sk_lock field of the sock structure. The lock field is defined as type socket_lock_t, in file *linux/include/linux/sock.h*. The field, slock, is the actual spinlock. Users is set to one when the socket is locked, set to zero when the socket is unlocked, and wq is the queue of waiting tasks.

```
typedef struct {
    spinlock_t          slock;
    unsigned int        users;
    wait_queue_head_t   wq;
}  socket_lock_t;
```

Generally, the lock is activated with the macro lock_sock.

```
lock_sock(sk)
```

The socket lock is released by calling the macro release_sock.

```
release_sock(sk)
```

Both macros are defined in *linux/include/linux/sock.h*. When the socket is locked, incoming packets are blocked from being put on the receive queue. Instead, they are placed in the backlog queue for later processing. This backlog queue and other aspects of receiving packets are described fully in Chapter 4.

5.12 IO System Calls and Sockets

Linux is a Unix-compatible operating system. Like other similar operating systems, Linux has a unified IO facility. All IO system calls are implementation independent; they work with devices, files, or sockets transparently. For most applications, no distinction is necessary. This has a major advantage in that the Linux application programmer does not have to remember three sets of API functions—one set for files, another for device IO, and a third for network protocols. This section discusses how file IO works with sockets and what is done in the socket layer itself to make this possible.

All IO devices, files, and other entities have a file descriptor. This file descriptor references an object in the file system called an inode, and all objects with file system-like behavior all have inodes. The inode allows the sockets to be associated with a Virtual File System (VFS). The inode structure is accessed with each IO system call such as read, write, fcntl, ioctl, and close. When an IO call is performed on an open socket, a pointer to the socket structure is retrieved from the inode. Remember that the open system call can't be used to create a socket; instead, sockets must be created with the socket API function. Once a socket is created, the IO calls work the same way with sockets as they do for files or devices.

The inode is created and initialized by the sock_alloc call defined in file *linux/net/socket.c*. As discussed earlier in Section 5.11, sock_alloc is called from sock_create. Sock_alloc actually creates both a socket structure and an inode structure from the socket inode slab cache. The online functions SOCKET_I and SOCK_INODE are provided *in linux/include/linux/socket.h* to map an inode to a socket and a socket to an inode, respectively.

```
static inline struct socket *SOCKET_I(struct inode *inode);
static inline struct inode *SOCK_INODE(struct socket *socket);
```

Once the inode is created, the socket layer can map the IO system calls. In *socket.c*, a file_operations structure, socket_file_ops, is created and initialized with pointers to socket versions of each of the IO system calls.

```
struct file_operations socket_file_ops = {
    owner:        THIS_MODULE,
    llseek:       no_llseek,
```

Llseek is set to the generic "no" version of the socket call because it is not supported for sockets, and the error return is OK.

```
    aio_read:     sock_aio_read,
    aio_write:    sock_aio_write,
    poll:         sock_poll,
    ioctl:        sock_ioctl,
    mmap:         sock_mmap,
```

The open call is not supported for sockets. Open is set to sock_no_open to disallow opening a socket via the /proc file system. It returns an ENXIO error.

```
    open:         sock_no_open,
```

```
    release:      sock_close,
    fasync:       sock_fasync,
    readv:        sock_readv,
    writev:       sock_writev,
    sendpage:     sock_sendpage
} ;
```

5.13 Netlink and Rtnetlink

Netlink is an internal communication protocol. It mainly exists to transmit and receive messages between the application layer and various protocols in the Linux kernel. Netlink is implemented as a protocol with its own address family, AF_NETLINK. It supports most of the socket API functions. Rtnetlink is a set of message extensions to the basic netlink protocol messages. The most common use of netlink is for applications to exchange routing information with the kernel's internal routing table.

Netlink sockets are accessed like any other sockets. Both socket calls and system IO calls will work with netlink sockets. For example, the sendmsg and recvmsg calls are generally used by user-level applications to add and delete routes. Both these calls pass a pointer to the nlmsghdr structure in the msg argument. This structure is defined in the file *linux/include/linux/netlink.h*.

```
struct nlmsghdr
{
```

Nlmsg_len is the length of the message including the header. The field, nlmsg_type, indicates the message content.

```
    __u32         nlmsg_len;
    __u16         nlmsg_type;
```

The next field, nlmsg_flags, are flags for the request. The values for nlmsg_flags are defined in Table 5.9. The next field, nlmsg_seq, is the sequence number for the message.

```
    __u16         nlmsg_flags;
    __u32         nlmsg_seq;
```

Table 5.9: Values for Netlink Message Flags

Name	Value	Purpose
NLM_F_REQUEST	1	This is a request message.
NLM_F_MULTI	2	This is a multipart message terminated by NLMSG_DONE.
NLM_F_ACK	4	The recipient should reply with an acknowledge, a zero, or an error code.
NLM_F_ECHO	8	The recipient should echo this request.
These values are modifiers to the netlink get request.		

Table 5.9: Values for Netlink Message Flags

Name	Value	Purpose
NLM_F_ROOT	0x100	Specify tree root.
NLM_F_MATCH	0x200	Return all matching entries.
NLM_F_ATOMIC	0x400	This is an atomic get.
NLM_F_DUMP	(NLM_F_ROOT| NLM_F_MATCH)	
These values are modifiers to the new request.		
NLM_F_REPLACE	0x100	Replace an existing entry.
NLM_F_EXCL	0x200	Check to see if an entry already exists.
NLM_F_CREATE	0x400	Create a new entry if it does not exist
NLM_F_APPEND	0x800	Add entry to the end of list.
Here we show the mapping of BSD 4.4 commands to the equivalent Linux netlink message flags values.		
BSD 4.4 Add	NLM_F_CREATE| NLM_F_EXC	Adds a new entry.
BSD 4.4 Change	NLM_F_REPLACE	Replace an existing entry.
BSD 4.4 Append	NLM_F_CREATE	Create a new entry if it does not exist.
BSD 4.4 Check	NLM_F_EXCL	Check to see if an entry already exists.

Nlmsg_pid is the sending Process Identification (PID) if the process is a user-level process, and zero if not.

```
    __u32         nlmsg_pid;
} ;
```

The netlink protocol is implemented in the file *linux/netlink/af_netlink.c*. It is like any other protocol in the TCP/IP protocol suite, except that it is for exchanging messages between user-level processes and internal kernel entities. It is similar to UDP or TCP in that it defines a proto_ops structure to bind internal calls with socket calls made through the AF_NETLINK address family sockets. The bindings are shown in the netlink_ops declaration also in the file *af_netlink.c*.

```
struct proto_ops netlink_ops = {
```

The address family type is PF_NETLINK.

```
    .family:      PF_NETLINK,
    .owner:       THIS_MODULE,
```

The protocol defines release, bind, and connect functions. Most of the other functions are not defined.

```
    .release=     netlink_release,
    .bind=        netlink_bind,
    .connect=     netlink_connect,
    .socketpair=  sock_no_socketpair,
    .accept=      sock_no_accept,
    .getname=     netlink_getname,
    .poll=        datagram_poll,
    .ioctl=       sock_no_ioctl,
    .listen=      sock_no_listen,
    .shutdown=    sock_no_shutdown,
    .setsockopt=  sock_no_setsockopt,
    .getsockopt=  sock_no_getsockopt,
```

Sendmsg and recvmsg are the main functions used to send and receive messages through AF_NETLINK sockets.

```
    .sendmsg=     netlink_sendmsg,
    .recvmsg=     netlink_recvmsg,
    .mmap=        sock_no_mmap,
    .sendpage=    sock_no_sendpage,
} ;
```

Just like other protocols, such as UDP and TCP that register with the socket layer, netlink address family declares a global instance of the net_proto_family structure in the file *af_netlink.c*.

```
struct net_proto_family netlink_family_ops = {
    .family = PF_NETLINK,
    .create = netlink_create,
    .owner = THIS_MODULE,
} ;
```

The netlink module also provides an initialization function for the protocol, netlink_proto_init.

```
static int __init netlink_proto_init(void);
```

This function registers the netlink family operations with the socket layer by calling sock_register.

5.13.1 Rtnetlink

In other chapters, we discuss how the routing tables and the neighbor cache are structured. In this section, we will show how the sendmsg and recvmsg functions are used with rtnetlink to pass requests for updates to the routing and the neighbor tables. All the operations break down to one of two fundamental operations: either retrieve the content of the table or post an update to the table. To support this, rtnetlink provides a structure defined in the file *linux/include/linux/rtnetlink.h*, called rtnetlink_link. This structure only contains only function pointers.

```
struct rtnetlink_link
{
    int (*doit)(struct sk_buff *, struct nlmsghdr*, void *attr);
    int (*dumpit)(struct sk_buff *, struct netlink_callback *cb);
} ;
```

Defined in the same file, rtnetlink also defines a global of instances of the preceding structure called rtnetlink_links. There are up to 32 of these instances, defined by NPROTO where each corresponds to a protocol type.

```
struct rtnetlink_link * rtnetlink_links[NPROTO];
```

To see how rtnetlink is used, let's look at an example. If a utility running at the application layer caller wants to add a route to a internal routing table, it calls recvmsg with a pointer to an nlmsghdr passed as an argument and with the nlmsg_type field is set to RTM_NEWROUTE. The socket layer will gather the message into an sk_buff structure and send to the netlink protocol, which in turn will queue it to the receive queue of a PF_NETLINK socket. The function rtnetlink_rcv_skb in file *rtnetlink.c* will get the message when it is de-queued from the socket.

```
extern __inline__ int rtnetlink_rcv_skb(struct sk_buff *skb);
```

In this function, the first thing we do is to get the nlmsghdr from the data part of the sk_buff and call the function rtnetlink_rcv_msg.

```
static __inline__ int rtnetlink_rcv_msg(struct sk_buff *skb,
struct nlmsghdr *nlh, int *errp);
```

Rtnetlink_rcv_msg gets nlmsg_type from nlh, and this value is used to call through the doit function pointer to add the route to the table. There is actually a little more to this process because rtnetlink uses an attached rtmsg structure that includes more information about the routing protocol so we know which routing table to access. There is more information about the aspects that are specific to IPv4 routing in Chapter 9, and IPv6 routing in Chapter 11.

5.14 Summary

This chapter was about Linux sockets and their implementation. We introduced sockets and discussed how Linux sockets are entirely compatible with the Posix standards and the socket implementations in other operating systems. We discussed how the socket layer is initialized and how the protocol families and their individual member protocols register with the socket layer. We discussed the socket API and included information about Linux special sockets. We also introduced Linux capabilities, and this is used to enforce restricted access to internal modifications.

In this chapter, we discussed the internal implementation of the socket layer and demonstrated what happens under the hood when one of the socket API functions is called from the application code. Finally, we discussed the netlink protocol and the rtnetlink extensions and how these are used to maintain the routing table and the destination cache.

Chapter 6: The Linux TCP/IP Stack

The first three chapters in this book covered basics of networking and a general background on TCP/IP implementations. As discussed in Chapter 1, "Introduction," each layer in a network protocol stack is independent from the layers above and the layers below. Each layer maintains its own logical relationship between the sending and receiving machines. Encapsulation is the primary means of maintaining the independence between the layers. However, in addition to encapsulation, the kernel must provide other facilities for TCP/IP to function. These facilities needed by the stack include timers, kernel threads or tasks, protocol registration facilities at the transport and network layers, a network driver interface, and a buffering scheme. Chapter 4, "Linux Networking Interfaces and Device Drivers," explored specifics of the network device driver interface, and Chapter 5, "Linux Sockets," discussed the socket API. This chapter builds on the material in Chapter 4 by exploring parts of the underlying structure of the Linux implementation of TCP/IP that don't fit neatly into later chapters on the proto -cols themselves. The facilities covered by this chapter include general caching and queuing facilities provided by the kernel so protocol implementations can function efficiently.

6.1 Introduction

Linux has evolved into an excellent and stable OS with readable well-commented source code. Although it has been improved with the 2.6 kernel, there are some inconsistencies in the naming conventions used in comments, variables, functions, and subsystems. Some of these inconsistencies are found in the parts of the stack covered in this chapter. Although they are actually separate facilities, the terms *bottom half*, kernel *tasklets*, and *softirqs* are often used interchangeably. Another example is that the network layer protocols are called *packet handlers*, but the registration facility for the packet handlers is sometimes referred to as *device registration*. Sometimes, the names of files are unrelated to their contents. For example, the file *linux/net/core/dev.c* doesn't just contain device-related information. It also contains the protocol management and registration functionality. Another example is the file *linux/include/linux/if_ether.h*, which contains the definitions for the protocol field in the Layer 2 headers for all Layer 2 headers, not just Ethernet.

There are strong similarities among TCP/IP implementations, and Linux reflects some of the implementation history in its names, conventions, and implementation choices. In this chapter, we examine the various queuing, hashing, and registration functions that provide the glue between the layers and protocols in TCP/IP as implemented in Linux. We will see how Linux has evolved into a complex system with the capability to add entire protocol families, and specific member protocol components. Protocol families can be implemented as modules or compiled into the kernel itself. We will see how the 2.6 version of the kernel has simplified some of the function's data structures and eliminated some of the redundant fields.

The TCP/IP protocol is complex and it is the subject of much study. It may seem that most of the complexity in the Linux TCP/IP stack implementation should be in the TCP and IP protocols themselves, but this is not entirely the case. Most modern OSs have a modular framework for networking. As discussed in Chapter 3, "TCP/IP in Embedded Systems," we must differentiate the network protocol from any particular implementation or implementation framework for the

protocol. With most modern network stack implementations, the framework is independent from the protocol. The network framework includes a device- independent interface between the network layer and the data link layer so multiple protocols can run simultaneously. The framework also provides a device-independent mechanism for network to physical address translation. Although Linux does not explicitly refer to its networking infrastructure as a "framework," it is there nonetheless. While other chapters talk about how the protocols work in Linux, in this chapter we will examine some components in the Linux networking framework.

6.2 Packet Handler Glue

Linux provides a method used by network and link layer protocols to dynamically register their input functions so they will receive input packets. Linux also provides registration mechanisms where the protocols at each layer can register with the layer above and the socket layer. Like other protocol stack implementations, Linux has to have a way for each layer to hide information and implementation details from the layers above and below. In this section, we discuss the glue that connects the network interface drivers to the network layer. We discuss how protocols receive packets from the network interfaces. Linux provides a dispatch mechanism so a network interface driver can pass an incoming packet up to the next layer based on the protocol number.

For example, after the data link layer, or Layer 2, is done processing incoming data, the packet must be efficiently dispatched to be handled by any protocol that has registered to receive this packet, such as IP, ARP, or IPv6. These de-multiplexing decisions are made based on the 2-byte protocol field in the link layer header. Table 6.1 shows most of the protocols in the 2-octet field in the protocol field in the data link layer, which are defined in the file *linux/include/linux/if_ether.h*. Some of the numbers in Table 6.1 are not official protocol numbers. For the official numbers, refer to the Internet Assigned Number Authority (IANA) [IAPROTO3].

Table 6.1: Link Layer Protocol Field Values

Protocol	**Value**	**Description**
ETH_P_LOOP	0x0060	Ethernet loopback packet
ETH_P_PUP	0x0200	Xerox PUP packet
ETH_P_PUPAT	0x0201	Xerox PUP Addr Trans packet
ETH_P_IP	0x0800	Internet Protocol
ETH_P_X25	0x0805	CCITT X.25
ETH_P_ARP	0x0806	Address Resolution Protocol (ARP)
ETH_P_BPQ	0x08FF	G8BPQ AX.25 Ethernet packet. This protocol number is not official.
ETH_P_IEEEPUP	0x0a00	Xerox IEEE802.3 PUP packet
ETH_P_IEEEPUPAT	0x0a01	Xerox IEEE802.3 PUP Addr Trans packet
ETH_P_DEC	0x6000	DEC assigned proto
ETH_P_DNA_DL	0x6001	DEC DNA Dump/Load
ETH_P_DNA_RC	0x6002	DEC DNA Remote Console

Table 6.1: Link Layer Protocol Field Values

Protocol	Value	Description
ETH_P_DNA_RT	0x6003	DEC DNA Routing
ETH_P_LAT	0x6004	DEC LAT
ETH_P_DIAG	0x6005	DEC Diagnostics
ETH_P_CUST	0x6006	DEC Customer use
ETH_P_SCA	0x6007	DEC Systems Comms Arch
ETH_P_RARP	0x8035	Reverse Addr Res packet
ETH_P_ATALK	0x809B	AppleTalk DDP
ETH_P_AARP	0x80F3	AppleTalk AARP
ETH_P_8021Q	0x8100	802.1Q VLAN extended header
ETH_P_IPX	0x8137	IPX over DIX
ETH_P_IPV6	0x86DD	IPv6
ETH_P_PPP_DISC	0x8863	PPPoE Discovery messages
ETH_P_PPP_SES	0x8864	PPPoE Session messages
ETH_P_ATMMPOA	0x884c	MultiProtocol over ATM
ETH_P_ATMFATE	0x8884	Frame-based ATM transport over Ethernet

The initialization of the Linux TCP/IP stack is a complex process, and part of this initialization is the protocol registration. The internal registration of the protocols of the stack really consists of three parts. The first part involves registering the network layer protocols (often called *packet handlers)* so they receive packets from the network interface drivers. The second part of the registration process is registering the transport layer protocols with the network layer, IP, so the appropriate transport layer such as UDP or TCP will be dispatched. The third part, which consists of registering the protocols with the socket layer, is discussed in detail in Chapter 5. Section 6.7 includes detailed information about transport layer protocol de-multiplexing and how the transport layer protocols tell IPv4 that they want packets of certain types. Chapter 4 contains a detailed discussion of the driver registration process, the network device structure, and the sequence of events that occur with packet reception and transmission. The next section discusses the registration of network protocols.

6.3 Linux TCP/IP Stack Initialization

Because of the structure of the Linux TCP/IP stack, it is difficult to separate basic TCP/IP initialization from the socket layer initialization. The initialization steps directly related to the socket layer are covered in Chapter 5, but the more general initialization mechanism of the TCP/IP protocol family is covered in this section. As discussed in Chapter 5, socket layer initialization occurs before the TCP/IP stack is initialized because the socket layer must be in place before the address family, AF_INET, can be registered with the socket layer, allowing any IP packets to be transmitted or received. Remember that AF_INET is another name for the TCP/IP protocol suite. It is used by the socket layer to direct the socket API calls to TCP/IP.

The meat of TCP/IP initialization is done by inet_init, which is defined in the file *linux/net/af_inet.c*. Chapter 5 describes the registration with the socket layer in detail. In function inet_init, after the socket registration is complete, we call the initialization functions for each of the protocols in TCP/IP. The first of these is arp_init, which we call to initialize the ARP protocol followed. Next, we call ip_init to initialize the IP protocol, followed by a call to tcp_init to set up the slab cache for TCP. This is followed by a call to icmp_init. If multicast routing is configured, its initialization function, ip_mr_init, is called next. See Chapter 9, Section 9.7 for more information on multicast routing.

One of the things done at initialization time is to create the entries in the /proc pseudo-file system for each of the protocols. Section 6.10.1 has more information about the /proc file system and other facilities provided by Linux for TCP/IP stack debugging.

6.3.1 Packet Handler Initialization and Registration

According to the OSI layered network model, a network layer protocol deals with the semantics of network addressing and packet routing. However, some OSs and TCP/IP stack implementations define a network layer protocol as anything that receives incoming packets from the network interface drivers. Linux defines all protocols that receive packets from network interface drivers as *packet handlers*. The protocol management and registration facility provided by Linux is called the Packet Handler Registration Facility. Sometimes, in Linux, we refer to the packet handlers as *taps*, and this term is probably more accurate because not all packet handlers do packet processing but just receive copies of incoming packets.

Linux provides two functions for registering and unregistering packet handlers. The registration function is called by a packet handler's initialization routine so it can set itself up to receive incoming packets. Registration is by type, so the packet handler will receive only incoming packets it is supposed to get. When packets arrive, they are de-multiplexed and dispatched to the packet handlers based on the type field of the link header. This de-multiplexing process is described later in Section 6.6.

The packet handler registration and unregistration functions and arguments are declared in the file, *linux/include/linux/netdevice.h***.** The functions are defined in file *linux/net/core/dev.c*. Both functions accept a pointer to a packet type structure, packet_type, as an argument.

```
struct packet_type {
```

The first field type is the same as the protocol type, which corresponds to the type field in the Ethernet header in network byte order. Table 6.1 shows the list of protocol types.

```
    unsigned short        type;
```

This field, dev, is a pointer to the net device structure for the network interface from which we want to receive the packet. If it is NULL, we will receive packets from any network interface.

```
    struct net_device     *dev;
```

The next field, func, points to the protocol's handler function for this protocol type. This function is called by the network queuing layer when the protocol type field is matched.

```
    int(*func) (struct sk_buff *, struct net_device *, struct packet_type *);
    void        *af_packet_priv;
```

List points to the list of packet handlers. This field is filled in when the packet handler is registered.

```
    struct list_head    list;
} ;
```

The first of the two functions, dev_add_pack, registers a packet handler.

```
void dev_add_pack (struct packet_type *pt);
```

The argument, pt, is a pointer to the packet_type structure earlier. The type field of pt is set to one of the packet types shown in Table 6.1. The type would be ETH_P_ALL, 0x0003, if the caller wants to receive all packets promiscuously. If the handler is promiscuous, pt is placed on the list of promiscuous handlers, pointed to by ptype_all; otherwise, pt is placed in the hash table, ptype_base.

The function dev_remove_pack unregisters a packet handler by removing the packet_type pointed to by pt from the registration list.

```
void dev_remove_pack(struct packet_type *pt);
```

Packet handlers are kept on a linked list because multiple protocols can register handlers to receive the same packet type. Since several protocol handlers may receive the same packet type, none of them should alter the packet contents without cloning them first. As we will see in Chapter 7, Linux provides packet copying and cloning functions so if a packet handler needs to modify the packet, it can do it on a safe copy. We need this because the same skb is passed to each protocol handler on the list in sequence. In multiple CPU implementations of Linux, several protocols registering for a particular packet_type, could receive the skb simultaneously while executing in different CPUs. However, the internal locking mechanisms in the kernel prevent any contention problems.

Not all handlers actually receive packets from the external world. Packets are routed internally for various reasons. The protocol type fields for an Ethernet packet are shown in Table 6.1 but Table 6.2 also lists pseudo-packet types used for internal packet routing. ETH_P_ALL is set in the packet_type structure if the handler wants to receive all packets no matter what their type. In most conventional cases, there will only be one protocol handler for each packet type.

Table 6.2: Nonofficial and Pseudo Protocol Types

Type	**Value**	**Description**
ETH_P_802_3	0x0001	Dummy type for 802.3 frames
ETH_P_AX25	0x0002	Dummy protocol ID for AX.25

Table 6.2: Nonofficial and Pseudo Protocol Types

Type	Value	Description
ETH_P_ALL	0x0003	Every packet. This type should be used with care.
ETH_P_802_2	0x0004	802.2 type framing
ETH_P_SNAP	0x0005	Internal only
ETH_P_DDCMP	0x0006	DEC DDCMP: Internal only
ETH_P_WAN_PPP	0x0007	Dummy type for WAN PPP frames
ETH_P_PPP_MP	0x0008	Dummy type for PPP MP frames
ETH_P_LOCALTALK	0x0009	Localtalk pseudo type
ETH_P_PPPTALK	0x0010	Dummy type for Atalk over PPP
ETH_P_TR_802_2	0x0011	802.2 frames
ETH_P_MOBITEX	0x0015	Mobitex (kaz@cafe.net)
ETH_P_CONTROL	0x0016	Card-specific control frames
ETH_P_IRDA	0x0017	Linux-IrDA
ETH_P_ECONET	0x0018	Acorn Econet

Each protocol or packet handler must have three components: a receive function, which is registered by calling dev_add_pack; an initialization function; and a declaration of the packet_type structure discussed previously. The protocol type field in the packet_type structure corresponds to the type field in the Ethernet packet or the type field in other MAC layer headers. Each protocol initializes an instance of the packet_type structure with the protocol number in the type field and a pointer to the packet handler's receive function in the func field. This initialization is usually done at compile time. The network protocol's initialization function does the registration of the handler.

The list of promiscuous packet handlers is pointed to by the static variable ptype_all in *linux/net/core/dev.c*.

```
static struct packet_type *ptype_all = NULL;
```

The linked list of handlers for specific protocol types is built as a 16-slot hash table for fast dispatch of the packet handler function.

```
static struct packet_type *ptype_base[16];
```

There is some overlap in the hash because there is a large number of defined protocol types and the hash has only 16 slots. However, the hashing function provides a much faster dispatch than could be obtained from a simple linear traversal of the list of packet types. Dispatching of each packet handler actually occurs within the context of the NET_RX_SOFTIRQ. The kernel threading mechanism, softirqs, and the specific kernel threads used for packet reception and transmission are discussed in detail in Section 6.4.2. Specifically, when the tasklet

NET_RX_SOFTIRQ is scheduled, it calls the function net_rx_action, which walks through the input packet queues and dispatches the packet hander functions.

Most of the packets we receive from the network interface drivers are IP packets. For an example, let's look the packet handler function for the IP protocol. IP is discussed in much more detail in Chapter 9, "The Network Layer, IP," but we in this section, we examine how the IP registers its packet handler. In file *net/ipv4/ip_output.c*, the IP protocol initializes ip_packet_type with the protocol type set to 0x0800 (IP) and initializes the func field to point to ip_rcv at compile time.

```
static struct packet_type ip_packet_type =
{
    type = __constant_htons(ETH_P_IP),
    func = ip_rcv,
} ;
```

The IP initialization function, also in the same file, is ip_init.

```
void __init ip_init(void)
{ . . .
```

This is how the ip packet_type structure is registered.

```
dev_add_pack(&ip_packet_type);. . .
```

6.4 Kernel Threading

In this book, we won't go into a theoretical discussion of multithreading models, nor will we examine the Linux scheduler design in detail. However, we will discuss how Linux kernel threads are used with TCP/IP. We will see that Linux provides a special thread to execute the registered packet handlers. This thread is separate from the network interface drivers' interrupt service routines. The thread runs concurrently with the interrupt handlers in the networking interface drivers. We know from studying real-time systems that interrupt handlers can consume a complete CPU. Moreover, even though packet handling within the stack is often faster than reading packets from the input device, we still need to minimize the time spent in interrupt handlers.

Especially with today's fast networks and network interface hardware, it is important to spend as little time as possible in the interrupt handler, or the whole system will bog down. In addition, TCP/IP, like other networking protocols, is asynchronous in nature. The amount of time needed to process a packet and hand it off to the application layer is dependent on the transport protocol and many other factors. Linux TCP/IP provides buffering at various layers so one component will not bog down the entire stack. Because of these factors, it is far preferable to allow the TCP/IP stack to process incoming packets independently from the driver. Some primitive TCP/IP implementations are intended for small footprint-embedded environments where memory utilization is more critical than processing time. These TCP/IP implementations complete the entire input packet processing in the context of the interrupt handler. However, modern operating systems used in all but the smallest embedded systems—and Linux is no

exception—provide multitasking OSs or at least several internal kernel threads or execution contexts.

6.4.1 The Bottom Line on the Bottom Half

Historically, Linux had a concept called the *bottom half*, which was responsible for all low-level processing. We will see how it got its name. Device drivers are generally thought of as consisting of a top part and a bottom part. The bottom part includes the interrupt handler, and the top part interfaces with the rest of the operating system. Earlier versions of the Linux kernel had a kernel threading facility called BH (*bottom half).* The name *bottom half* was chosen because it is analogous to the bottom part of a device driver, its interrupt service routine. The bottom half in earlier versions of the Linux kernel was a lightweight pseudo-interrupt that does work deferred from hardware IRQs after re-enabling interrupts. The bottom half typically does work that can be done while interrupts are re-enabled but can't wait for a heavier kernel task or user-level process to get scheduled. Earlier versions of Linux relied exclusively on bottom halves to provide kernel threads for TCP/IP packet processing. However, now we need a more complex model to describe the bottom halves, because recent versions of Linux are designed to run on multiple CPU hardware. Linux is Symmetric Multiprocessing (SMP) aware, so a group of CPUs can share the same memory space. The old bottom-half mechanism was not SMP aware and had to go. This is because the bottom-half mechanism would hold off the other CPUs in a multi-CPU computer. When the bottom half was executing, all CPUs other than the one running the BH must be stopped. In the next section, we will see how tasklets and softIRQs solve this problem in recent versions of Linux. For an excellent discussion of Linux kernel threading, see Matthew Wilcox's paper presented at Linux Conference Australia in 2003 [WILC03].

6.4.2 Tasklets, SoftIRQs, and Timers

Newer versions of Linux include a multithreading facility called tasklets. Bottom halves still exist, but they have been converted into tasklets, which are SMP aware. In addition to tasklets, the Linux kernel also provides timers for use by device drivers and protocols. The discussion on kernel threads in this book is limited to the threading model used internally in the TCP/IP stack and the network interface drivers. However, Bovet and Cesati include a complete discussion of Linux kernel threads [BOVET02]. In addition, for a complete examination of the art of using Linux kernel threads in device drivers, see [RUBINO00].

The general Linux kernel threading facility is called softIRQs. Linux also has special softIRQs that only execute other kernel threads called tasklets. Theoretically, it is possible to have up to 32 softIRQs in the kernel. However, Linux does not provide an interface for dynamically creating softIRQs directly, and it is not possible to add a softIRQ without modifying the source. However, adding more softIRQs to the kernel is neither necessary nor encouraged for the network protocol or device driver developer. Instead, the kernel provides tasklets that can be allocated dynamically, and Linux provides a predefined softIRQ to run the tasklets. The tasklet facility has its own set of interface functions and is quite sufficient for most kernel threading requirements needed by both protocol and driver developers.

One of the predefined softIRQs is specifically for executing special high-priority tasklets. Two of these softIRQs are used by the queuing layer in TCP/IP. One of the softIRQs is for receive-side

packet processing, and the other softIRQ is for transmit-side packet processing. Table 6.3 below lists each type of softIRQ.

Table 6.3: SoftIRQs

SoftIRQ	**Purpose**
HI_SOFTIRQ	For high-priority tasklets.
NET_TX_SOFTIRQ	Transmission-side packet processing from link layer and network layer protocols. This one is used by TCP/IP.
NET_RX_SOFTIRQ	Reception-side packet processing from network interface drivers, and this one is used by TCP/IP, too.
TASKLET_SOFTIRQ	For running tasklets.

Tasklets are ideal for implementing minijobs to do high-priority work that is deferred from device driver ISRs but can be done with interrupts enabled. Tasklets are used widely in the TCP/IP stack. For example, they are used by the neighbor cache facility, which is a device—and protocol—independent address translation facility to map network layer IP addresses to link layer MAC addresses. Tasklets are defined by the structure tasklet_struct, which is declared in the file *linux/include/linux/interrupt.h*.

```
struct tasklet_struct
{
```

The field, next, is so the tasklet_struct instances can be put in a list.

```
    struct tasklet_struct *next;
```

State is the scheduling state of the tasklet.

```
    unsigned long state;
```

The next field, count, must be equal to zero for the tasklet to be scheduled.

```
    atomic_t count;
```

Func points to the routine that is executed when the tasklet is scheduled.

```
    void (*func)(unsigned long);
    unsigned long data;
} ;
```

The interface functions for controlling the operation of tasklets is also in the file *linux/include/linux/interrupt.h*. The first of these functions is tasklet_init, which initializes the tasklet_struct instance pointed to by t.

```
void tasklet_init(struct tasklet_struct *t, void(*func)(unsigned long),
unsigned long data);
```

The next function is tasklet_kill, which we call when we want to stop the tasklet from executing.

```
void tasklet_kill(struct tasklet_struct *t);
```

The function tasklet_schedule prepares a tasklet for scheduling.

```
void tasklet_schedule(struct tasklet_struct *t);
```

Tasklet_disable prevents a tasklet from being scheduled by incrementing the count field in the structure pointed to by t.

```
void tasklet_disable(struct tasklet_struct *t);
```

Finally, the last function in this group, tasklet_enable allows a tasklet to be scheduled by decrementing the count field in tasklet_struct.

```
void tasklet_enable(struct tasklet_struct *t);
```

There are several macros provided to aid in establishing tasklets. The first, DECLARE_TASKLET, fills in the fields of the tasklet structure.

```
DECLARE_TASKLET(name, func, data)
```

The macro DECLARE_TASKLET_DISABLED declares a tasklet but leaves it disabled by initializing the count to zero.

A third kernel facility, called *timers*, is used by TCP/IP. Timers are not threads. They run at interrupt level, so the timer functions should not do any elaborate processing. If more extensive processing is required, it should be done in a tasklet because tasklets can be pre-empted by interrupts. Linux timers are dynamic—they can be added or deleted at any time. The timer API functions are defined in *linux/include/timer.h*. A timer is created with a timer_list structure.

```
struct timer_list {
```

The first field, entry, is used internally for maintaining the list of timers.

```
    struct list_head entry;
```

The next field, expires, is the time when the timer goes off in ticks.

```
    unsigned long expires;
    spinlock_t lock;
    unsigned long magic;
```

The function, field points to the routine that is executed when the timer goes off.

```
    void (*function)(unsigned long);
```

The value in the next field, data, is passed as an argument to function when it is called.

```
    unsigned long data;
    struct tvec_t_base_s *base;
} ;
```

A kernel module may define a timer by declaring a static timer_list structure, and initializing a few fields. This is usually done with a macro TIMER_INITIALIZER also defined in file *linux/include/linux/timer.h*.

```
TIMER_INITIALIZER(_function, _expires, _data)
```

The API for Linux timers is fairly straightforward. It includes functions for initial izing timers, adding them and deleting them from the list of timers, and resetting the time when they go off. All of these function definitions are also found in the file, *linux/include/linux/timer.h*. The first interface function, init_timer, is provided to initialize the next and previous pointers in the list field in the timer_list to NULL.

```
static inline void init_timer(struct timer_list * timer);
```

The function add_timer activates the timer by adding it to the list of timers to be scheduled. Init_timer should be called before add_timer is called.

```
void add_timer(struct timer_list * timer);
```

This function, del_timer, removes the timer from the list. It returns a zero if the timer was pending and a one if the timer was not pending.

```
int del_timer(struct timer_list * timer);
```

Another function, mod_timer, updates the expires field of an active timer.

```
int mod_timer(struct timer_list *timer, unsigned long expires);
```

It returns zero if the timer was pending. If the timer was not pending, it creates a new timer using the value in expires and returns the value one.

Last but not least, Linux maintains the current time in a global variable, which is used all over the kernel. The variable is called jiffies and contains the current time in ticks.

6.5 Packet Queuing Layer and Queuing Disciplines

In this section, we examine one part of the Linux framework for networking, the *queuing layer*. Most of the functionality of the packet queuing layer is implemented in the file *linux/net/core/dev.c* and declared in *linux/include/linux/netdevice.h*.

We will see how packets are sent to the network interface driver from the network layer protocols. The primary purpose for the queuing layer is to provide independence and buffering between the device drivers and the network layer protocols. Another purpose is traffic shaping. To perform traffic shaping, multiple queue disciplines are provided to allow packet transmission decisions to be based on Quality of Service (QoS) and other factors. For a complete explanation

of practical applications for Linux traffic shaping, see *Linux Advanced Routing & Traffic Control HOWTO*, Chapter 9, "Queuing Disciplines for Bandwidth Management" [HUBHOW02]. The queuing layer is actually a separate entity from both the network interfaces and IP; it is a sliver between the device drivers and the network layer protocols. Although predominantly for transmission, the queuing layer includes facilities for both packet reception and transmission.

IP transmits the packet when it is done processing. This is after it has made the routing decision and done any necessary fragmentation and filled in the fields of the IP header. At this point, we might think there would be a simple handoff to the network interface driver to transmit the packet. In addition, on the input side, we might think that once a device driver completes the reception of an incoming packet, it should be a simple matter to pass the packet to IP's input route. However, things are quite a bit more involved. The queuing layer is the piece of framework that lives between the network layer protocols or packet handlers and the network interface drivers. The queuing layer has many responsibilities. On the input side, it starts and manages the softIRQs that does the processing of input packets received from the network interface driver. On the output side, the queuing layer takes over the transmission from the network layer protocols' output functions once they are ready to transmit the packet. This layer also contains the infrastructure to support multiple queue disciplines and multiple traffic-based scheduling methods for packet transmission. This is how Linux provides traffic shaping capability.

Linux version 2.2 introduced the softIRQs, which provide kernel threads that provide a separate context for protocol handlers to execute independently from the drivers' interrupt service routines. In contrast, earlier versions of the Linux kernel, as is the case with many other OSs, used with embedded systems, almost the entire input packet processing is done in the driver's interrupt context. In this section, we discuss the input side of the packet queuing layer and how it gets an incoming packet from the network interface driver's receive-side interrupt handler. We also discuss how the queuing layer handles transmitted packets. To facilitate traffic shaping, the transmit side of the packet queuing layer includes multiple queuing disciplines and multiple traffic classes when making decisions about transmitting a packet. We will examine how this is done.

As discussed earlier, Linux has two preconfigured softIRQs that provide the execution context for the most of the TCP/IP stack. One of these threads is the receive-side thread, NET_RX_SOFTIRQ. This softIRQ continues the packet processing after the network interface driver's receive interrupt is done. NET_RX_SOFTIRQ handles queuing and multiplexing of received packets. It schedules the receive-side functions of all the protocols that have registered to receive packets. The receive-side thread decides which of the registered protocols will receive an input packet, handles input packet queues, and determines whether there is internal network congestion. In Chapter 4, we learned how a network interface driver's ISR calls netif_rx to queue the input packet for processing by the network layer protocols. Netif_rx is called by the ISR so it must run quickly. Essentially, it queues up the packet and starts the receive softIRQ thread, NET_RX_SOFTIRQ, by calling cpu_raise_softIRQ, which causes the softIRQ's action function net_rx_action to be scheduled.

The other softIRQ, NET_TX_SOFTIRQ, handles packet transmission once the network protocols' output functions are ready to hand off packets for transmission. This softIRQ handles

flow control, packet queuing, and multiple queue disciplines. It also does the network interface driver level de-multiplexing to determine which driver should get the output packet.

The queuing layer provides congestion handling and flow control for received packets. It reschedules the protocol's input function if the protocol is too busy to receive a packet when the network interface driver's interrupt handler has it ready. Essentially, this is a layer of insulation between the protocol's input function such as IP and the network device driver. As discussed earlier, our goal is to offload processing from the interrupt service routines in the network device drivers. If it is not possible to re-queue packets, they will be dropped. However, in most cases, the packet is simply either put on the input queue, which generally has enough elasticity to absorb input packets waiting for processing.

6.5.1 Input Packet Queues, the Softnet_data Structure

The packet queuing layer has the capability to handle both hard device drivers with interrupt service routines and soft drivers, such as Layer 2, bridging code or packet forwarding engines. The packet queuing layer consists of an internal data structure called softnet_data.

```
struct softnet_data
{
```

The first field, throttle, is used for congestion control. When congestion is indicated and throttle is set, the incoming packet is dropped.

```
    int                 throttle;
```

Cng_level is the congestion level returned by the netif_rx function. This function is discussed in detail in Chapter 4. The next field, avg_blog, is the average backlog.

```
    int                 cng_level;
    int                 avg_blog;
```

Softnet_data contains pointers to both reception and transmission queues. The input_pkt_queue field is for handling received packets. It points to the head of a list of packets or sk_buffs ready to be passed up to the network layer protocols. Packets are put in this queue by the netif_rx function.

```
    struct sk_buff_head  input_pkt_queue;
```

Poll_list is a list of network interfaces to be processed. It is used by backlog processing.

```
    struct list_head     poll_list;
```

Output_queue is a list of net_device structures for network interface drivers ready to transmit packets. The output queue works with any queue discipline that is installed in Linux's list of queue disciplines for the particular driver.

```
    struct net_device   *output_queue;
    struct sk_buff      *completion_queue;
```

A pseudo net_device structure is used for queue maintenance; only a few flags and fields are used.

```
    struct net_device   backlog_dev;
} ;
```

Softnet_data instances are in an array indexed by CPU number to support separate queues on each CPU for SMP configurations. In addition to the softnet_data structure, the queuing layer includes several queue processing functions that are independent of the queue discipline installed for a particular network interface driver, and these functions are described in detail in Chapter 4.

6.5.2 Queuing Layer Initialization

The queuing layer is initialized by the function net_dev_init, defined in the file *linux/net/core/dev.c*.

```
static int __init net_dev_init(void)
{
int i, rc = -ENOMEM;
```

Net_dev_init is called from the network registration function when the very first network interface device is registered. For more details about network interface device registration, see Chapter 4. It makes sure that it is only called once by checking the static variable dev_boot_phase as soon as it begins executing. Dev_boot_phase is reset when the initialization is completed.

```
    BUG_ON(!dev_boot_phase);
    if (dev_proc_init())
        goto out;
    if (netdev_sysfs_init())
        goto out;
```

We initialize the list of promiscuous packet handlers.

```
    INIT_LIST_HEAD(&ptype_all);
    for (i = 0; i < 16; i++)
        INIT_LIST_HEAD(&ptype_base[i]);
```

We initialize the input packet queue for each CPU.

```
    for (i = 0; i < NR_CPUS; i++) {
        struct softnet_data *queue;
        queue = &per_cpu(softnet_data, i);
        skb_queue_head_init(&queue->input_pkt_queue);
```

The throttle flag and cng_level are set to zero and the average backlog.

```
        queue->throttle = 0;
        queue->cng_level = 0;
```

Avg_blog field is set to an arbitrary nonzero value, and the completion_queue is set to NULL.

```
        queue->avg_blog = 10;
        queue->completion_queue = NULL;
```

Next, we initialize the poll_list.

```
        INIT_LIST_HEAD(&queue->poll_list);
        set_bit(__LINK_STATE_START, &queue->backlog_dev.state);
        queue->backlog_dev.weight = weight_p;
        queue->backlog_dev.poll = process_backlog;
        atomic_set(&queue->backlog_dev.refcnt, 1);
    }
#ifdef OFFLINE_SAMPLE
    samp_timer.expires = jiffies + (10 * HZ);
    add_timer(&samp_timer);
#endif
    dev_boot_phase = 0;
```

Now, we start up the two softIRQs.

```
    open_softIRQ(NET_TX_SOFTIRQ, net_tx_action, NULL);
    open_softIRQ(NET_RX_SOFTIRQ, net_rx_action, NULL);
    dst_init();
    dev_mcast_init();
#ifdef CONFIG_NET_SCHED
    pktsched_init();
#endif
    rc = 0;
out:
    return rc;
}
```

The primary function of net_dev_init is to initialize the softnet_data structure c. In addition, a few fields in the backlog device (blog_dev) part of the softnet_data structure are initialized, and state is set to START to indicate that the queue is ready to accept packets. The field weight is set to the value 64, and the poll field is set to point to the process_backlog, which gets called later to remove the packets from the backlog queue. Finally, the refcnt field in the blog_dev is atomically set to one.

6.5.3 Queuing Transmitted Packets

In this section, we discuss how an output packet is prepared for transmission through the network interface driver. For an example, we will use the output function in the IPv4 protocol, ip_output, defined in the file *linux/net/ipv4/ip_output.c*.

```
int ip_output(struct sk_buff *skb);
```

Before it can transmit a packet, the IP protocol must build the IP header, fill in the destination address, and determine the next-hop recipient for an output packet. When it is ready to transmit, the output function is called with a pointer to a socket buffer containing the packet. We will not describe everything IPv4 does to ready a packet for transmission. Refer to Chapter 9 for details about the IPv4 protocol implementation in Linux. In this section, we focus on the last few steps performed by IPv4output function ip_output. The mechanism we describe shows how the IPv4

protocol or any other network layer protocol transmits a packet by de-referencing generic pointers in the destination cache so it does not need to know how the packet is going to be transmitted or whether it is transmitted at all.

The output function looks at the destination cache entry for this packet to see if it is supposed to transmit the packet or if the packet is to be sent to an internal destination. We check for a valid destination by looking at the destination cache entry to see if it is a nonzero value. If there is an entry in the destination cache, we check to see if the destination is resolved. We know it is resolved if there is a hardware header cache entry, hh. If hh is not zero, we call the function pointed to by the hh_output field of the hh_cache entry with a pointer to the sk_buff. At this point, the sk_buff holds the packet ready for transmission.

For destination cache entries that resolve to external destinations, hh_output will be set to dev_queue_xmit, and this is the function that we call from IP to send the packet.

```
int dev_queue_xmit(struct sk_buff *skb);
```

First, we check to see if the device has a registered queue discipline. Most drivers for Ethernet devices and other conventional hardware network interfaces use output queuing. However, pseudo-devices—devices without an interrupt handler—such as IP tunnels, the loopback device, or other specialized Layer 2 interfaces don't use a queue discipline. The queue discipline is determined by checking the qdisc field in the net device structure to see if it is initialized. If the device has a queue discipline, the enqueue function for the queue discipline associated with the device is called. For Ethernet devices, the default queue discipline is pfifo_fast. For pseudo-devices—devices that don't have an interrupt service routine—the default queue discipline is set to none. The packet gets placed on the end of the queue. Once the packet is queued, dev_queue_xmit will execute when the queue is scheduled. Dev_queue_xmit calls qdisc_restart to remove the packet from the queue and check to see if the driver's transmitter is busy. If it is busy, the packet is re-queued. If not, the driver's hard_start_xmit service function is called. For an illustration of the flow of the transmitted packet, see Figure 6.1.

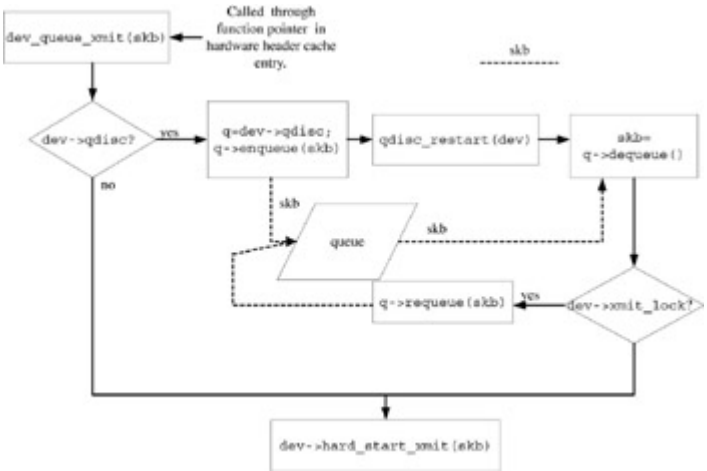

Figure 6.1: Transmit packet sequence.

6.6 Receiving Packets in Packet Queuing Layer, NET_RX_SOFTIRQ

From earlier discussions (Chapter 4, Section 4.4), we know that each network interface driver's interrupt service routine calls a generic function to queue up incoming packets for further processing. We recall from Section 6.4 that Linux TCP/IP provides a softIRQ kernel thread, NET_RX_SOFTIRQ, to process the queued up input packets. Actually, the kernel thread (softIRQ) does most of the packet processing in the TCP/IP stack, including the transport protocols UDP and TCP. In general, the purpose of this thread is to remove each packet from the input packet queue and execute the packet handling function for any packet handler whose type matches the protocol field in the link layer packet header. The packet handlers can be any protocol modules that expect to receive an incoming packet and have registered with the packet queuing layer (see Figure 6.2).

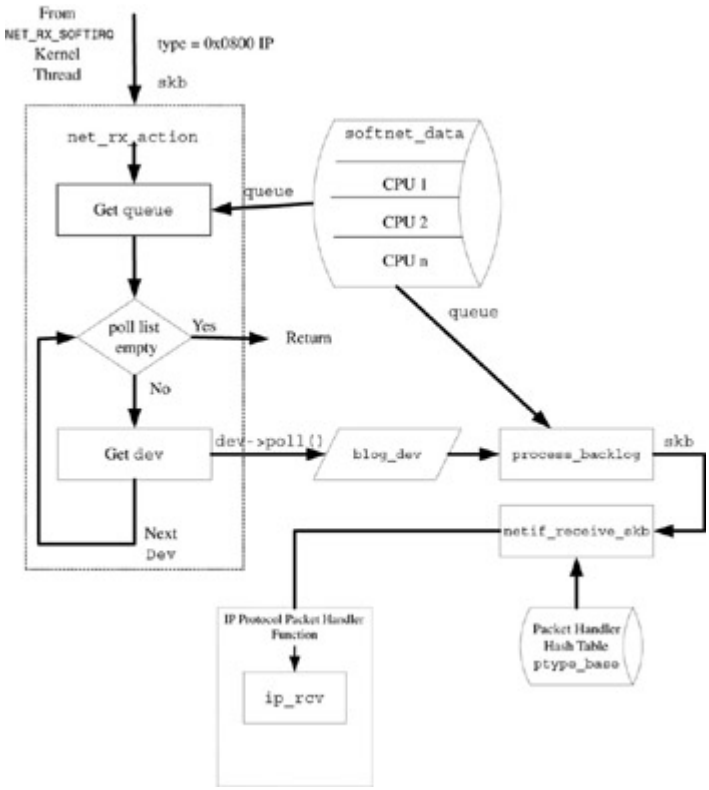

Figure 6.2: Receive packet sequence.

As we know from Section 6.4, each softIRQ has an action function. The action function associated with the network receive thread, NET_RX_SOFTIRQ, is net_rx_action, defined in file *linux/net/core/dev.c*. The essential activity of this function is to process the packets on the input packet queue, pointed to by the input_pkt_queue field of the softnet_data structure described earlier. As discussed in Chapter 4, packets were put on the input packet queue earlier when each network driver calls the netif_rx function. It is the job of the function net_rx_action to remove the packets from the queue and call the packet handlers. However, it does not actually remove the packets directly. Instead, it calls each driver's poll function through the poll field in the net_device structure for the input network interface from which the packets came. Actually, most networking device drivers don't have poll functions. Usually, poll functions are used with

pseudo-drivers that don’t have interrupts. However, we also have a "fake" device called the backlog device, backlog_dev. As shown earlier, backlog_dev is initialized when the softnet_data is created, and its poll function, process_backlog, does most of the "real" work of removing the input packets from the queue and calling the various protocols’ packet handling routines. Net_rx_action runs in the context of the NET_RX_SOFTIRQ. Net_rx_action walks through the poll list of all the attached input network interfaces and executes the function pointed to by the poll field in the net_device structure for each driver if there is one.

```
static void net_rx_action(struct softIRQ_action *h)
{
```

The softnet_data structures are actually in an array indexed by CPU number for SMP configurations, and get_cpu_var returns the queue for the correct CPU.

```
    struct softnet_data *queue = &__get_cpu_var(softnet_data);
    unsigned long start_time = jiffies;
```

Netdev_max_backlog is a sysctl constant that determines the maximum number of packets that can be queued up.

```
    int budget = netdev_max_backlog;
    preempt_disable();
    local_irq_disable();
```

Here we get each interface driver on the poll_list. One of these will be the pseudo- device, backlog_dev.

```
    while (!list_empty(&queue->poll_list)) {
        struct net_device *dev;
        if (budget <= 0 || jiffies - start_time > 1)
            goto softnet_break;
        local_irq_enable();
        dev = list_entry(queue->poll_list.next, struct net_device,
poll_list);
```

Here is where we call poll functions for each network interface driver, dev. If dev points to the backlog pseudo-device, backlog_dev, process_backlog, is the function we actually call through the poll pointer. The poll function returns zero if it successfully processed all the packets on the input queue and returns a negative one if the quota was exceeded.

```
        if (dev->quota <= 0 || dev->poll(dev, &budget)) {
            local_irq_disable();
            list_del(&dev->poll_list);
            list_add_tail(&dev->poll_list, &queue->poll_list);
            if (dev->quota < 0)
                dev->quota += dev->weight;
            else
                dev->quota = dev->weight;
        }  else {
            dev_put(dev);
            local_irq_disable();
        }
```

```
    }
out:
    local_irq_enable();
    br_read_unlock(BR_NETPROTO_LOCK);
    return;
softnet_break:
    __get_cpu_var(netdev_rx_stat).time_squeeze++;
    __raise_softIRQ_irqoff(NET_RX_SOFTIRQ);
}
```

6.6.1 The Process_backlog Function

As discussed earlier, the backlog device provides flow control for incoming packet processing, and the packet input processing posts incoming packets to the input packet queue in the backlog device. The poll field of the backlog device points to the function that does the next step of processing the input packets. This function is process_backlog.

```
static int process_backlog(struct net_device *backlog_dev, int *budget)
{
    int work = 0;
    int quota = min(blog_dev->quota, *budget);
```

We must get the queue instance from softnet_data for the current CPU. This will point to a list of devices containing queues of packets for processing.

```
    struct softnet_data *queue = &__get_cpu_var(softnet_data);
    unsigned long start_time = jiffies;
```

This function loops until all the packets on the backlog devices input packet queue are processed. However, it will not entirely take over the CPU in which it is running if there are a large number of packets to process. Therefore, the number of packets to process is controlled by the parameter budget. This mechanism will help prevent the kernel from robbing all the bandwidth in the current CPU from other high- priority tasks.

```
    for (;;) {
        struct sk_buff *skb;
        struct net_device *dev;
```

We must disable hardware interrupts to protect the input_pkt_queue because the packets were put on this queue by network interface drivers' ISRs.

```
        local_irq_disable();
        skb = __skb_dequeue(&queue->input_pkt_queue);
        if (!skb)
            goto job_done;
        local_irq_enable();
```

Here we get the "real" interface device from the skb and remove the device from the queue. Netif_receive_skb is called to process the receive packet.

```
        dev = skb->dev;
        netif_receive_skb(skb);
```

```
        dev_put(dev);
        work++;
        if (work >= quota || jiffies - start_time > 1)
            break;
```

If network hardware flow control is configured, netdev_wakeup is called to control the activity of network device drivers registered with the flow control mechanism.

```
#ifdef CONFIG_NET_HW_FLOWCONTROL
        if (queue->throttle && queue->input_pkt_queue.qlen <
no_cong_thresh ) {
            queue->throttle = 0;
            if (atomic_dec_and_test(&netdev_dropping)) {
                netdev_wakeup();
                break;
            }
        }
#endif
    }
```

We drop through to this point if we run out of time before we process all the input packets. A nonzero return tells the caller, net_rx_action, that we didn't finish our work. The caller's budget is adjusted.

```
    blog_dev->quota -= work;
    *budget -= work;
    return -1;
```

We jump here once we have successfully processed all packets on the queue.

```
job_done:
    blog_dev->quota -= work;
    *budget -= work;
    list_del(&blog_dev->poll_list);
    smp_mb__before_clear_bit();
    netif_poll_enable(backlog_dev);
    if (queue->throttle) {
        queue->throttle = 0;
#ifdef CONFIG_NET_HW_FLOWCONTROL
        if (atomic_dec_and_test(&netdev_dropping))
            netdev_wakeup();
#endif
    }
```

We re-enable interrupts before we exit.

```
    local_irq_enable();
    return 0;
}
```

6.6.2 The netif_receive_skb Function

The next function we discuss is called from the process_backlog, described in the previous section. Netif_receive_skb processes each packet on the input packet queue.

```
int netif_receive_skb(struct sk_buff *skb)
{
```

Ptype and pt_prev are used to go through the list of packet_type structures. These list entries point to network layer protocols that have registered receive handler functions with the queuing layer.

```
    struct packet_type *ptype, *pt_prev;
    int ret = NET_RX_DROP;
```

Type is set to the value of the protocol field in the skb, which contains the protocol number from the link layer header. We set the arrival time for each incoming packet in the stamp field of the skb. The receive statistics are also incremented.

```
    unsigned short type = skb->protocol;
    if (!skb->stamp.tv_sec)
        do_gettimeofday(&skb->stamp);

    skb_bond(skb);

        __get_cpu_var(netdev_rx_stat).total++;
```

CONFIG_NET_FASTROUTE configures fast routing. It is for quick processing of packets that go directly from incoming interface to outgoing interface bypassing most processing done by IPv4. If this option is set and the packet type indicates that it is to be fast routed, the packet is immediately transmitted without further processing. See Chapter 9 for more details about the IP routing implementation.

```
#ifdef CONFIG_NET_FASTROUTE
    if (skb->pkt_type == PACKET_FASTROUTE) {
    __get_cpu_var(netdev_rx_stat).fastroute_deferred_out++;
        return dev_queue_xmit(skb);
    }
#endif
```

At this point, we are in an early stage of packet processing; network layer processing hasn't happened yet and the skb still contains the packet headers. Therefore, the skb field data points to the beginning of the header information so we update the raw header pointers to point to the header start.

```
    skb->h.raw = skb->nh.raw = skb->data;
    pt_prev = NULL;
```

This loop is for promiscuous packet handling. The variable ptype_all is the start of the list of protocols that have requested promiscuous packet reception.

```
    list_for_each_entry_rcu(ptype, &ptype_all, list)
        if (!ptype->dev || ptype->dev == skb->dev) {
            if (pt_prev)
```

Deliver_skb updates the use count in the skb and calls the protocol's packet handler function through the func field of pt_prev.

```
                ret = deliver_skb(skb, pt_prev, 0);
            pt_prev = ptype;
        }
    }
```

Linux has an option called the frame diverter to allow for fast packet handling at this layer before packets are passed to the protocols for processing. The function handle_diverter will check to see if the packet skb should be diverted.

```
    handle_diverter(skb);
```

Next, we see if bridging has been implemented. Remember from earlier chapters that bridging is fundamentally a packet copy at Layer 2. Bridging consumes the incoming packet, so if it is bridged, we have no need for further processing.

```
    if (__handle_bridge(skb, &pt_prev, &ret))
        goto out;
```

Here we find the protocols that have registered to receive the packets of a specific type, such as IPv4 packets, where the type is 0x800. We use the same function we used for promiscuous handlers earlier in this function. However, in this case, deliver_skb is called for each packet_type instance on the list of registered handlers that matches the protocol in the incoming packet. For example, the IP protocol's registered packet handler function is ip_rcv, since it is set in the func field of the packet handler structure for packets having the type 0x0800.

```
    for list_for_each_entry_rcu(ptype, &ptype_base[ntohs(type)&15], list) {
        if (ptype->type == type &&
              (!ptype->dev || ptype->dev == skb->dev)) {
            if (pt_prev)
```

This is where we call the protocol's packet handler function.

```
                ret = deliver_skb(skb, pt_prev, 0);
            pt_prev = ptype;
        }
    }
    if (pt_prev) {
        ret = pt_prev->func(skb, skb->dev, pt_prev);
    }  else {
        ret = pt_prev->func(skb, skb->dev, pt_prev);
        kfree_skb(skb);
        ret = NET_RX_DROP;
    }
out:
    rcu_read_unlock();
    return ret;
}
```

6.7 Transport Layer De-Multiplexing and Internal Packet Routing

Transport layer de-multiplexing is the process of deciding which transport layer protocol will receive an incoming IP packet when IP processing is complete. When IPv4 is done with a received packet, it must pass the protocol up to a transport layer protocol or one of the protocols internal to IPv4 such as ICMP and IGMP. The protocol field in the IP header determines which member protocol in the AF_INET family will receive the packet. Each of the protocols registered at initialization time to receive the packets of the type specified by the 1-byte protocol field in the IP header. We recall that the protocol field is not the same as the protocol field in the Ethernet MAC header. Link layer and IPv4 Header encapsulation is discussed in Chapter 3.

6.7.1 The inet_protos Array

The protocols are registered by the inet_init function when the kernel is initialized in the file *linux/net/ipv4/af_inet.c*. As part of this initialization, pointers to each of the protocols are placed in an array of pointers called inet_protos This array is declared in the file *linux/net/ipv4/protocol.c*.

```
struct inet_protocol *inet_protos[MAX_INET_PROTOS];
```

Each protocol declares an instance of the structure inet_protocol.

```
struct inet_protocol
{
```

Handler points to the protocol's input function, and err_handler points to the protocol's error function if one is defined. Later, we show how these fields are initialized for the higher-layer protocols in TCP/IP.

```
    int             (*handler)(struct sk_buff *skb);
    void            (*err_handler)(struct sk_buff *skb, u32 info);
```

The last field has no use in IPv4. It corresponds to a field in IPv6 that determines the security association policy for the protocol.

```
    int             no_policy;
} ;
```

Each protocol is placed in the hash array called inet_protos. The array can hold up to up to 255 protocols. Four locations in the array are filled when the four protocols are added that are the basic part of the IPv4 protocol suite: UDP, TCP, ICMP, and IGMP. This is done during AF_INET family initialization. Protocols are put in the inet_protos hash table by calling inet_add_protocol. After initialization, protocols can be added dynamically.

This is how inet_protocol instances are initialized for the three permanent protocols in the IPv4 protocol suite: TCP, UDP, and ICMP. IGMP is shown also, because this protocol is configured if multicast routing is configured into Linux at kernel build time.

```
#ifdef CONFIG_IP_MULTICAST
static struct inet_protocol igmp_protocol = {
    handler =    igmp_rcv,
} ;
#endif
static struct inet_protocol tcp_protocol = {
    handler =    tcp_v4_rcv,
    err_handler =tcp_v4_err,
    no_policy =  1,
} ;
static struct inet_protocol udp_protocol = {
    handler =    udp_rcv,
    err_handler =udp_err,
    no_policy =  1,
} ;
static struct inet_protocol icmp_protocol = {
    handler =    icmp_rcv,
} ;
```

6.7.2 Adding and Removing Protocols from the Inet_protos Array

Two functions, defined in file *linux/include/net/protocol.h*, are provided by Linux to add or delete protocols from the hash table. The first of these functions is net_add_protocol, which registers a new protocol by adding prot to the hash table of protocols, inet_protos.

```
void inet_add_protocol(struct inet_protocol *prot);
```

Inet_del_protocol removes prot from the hash table.

```
int inet_del_protocol(struct inet_protocol *prot);
```

The protocols are accessed with a 1-byte hash index, and since there is only a 1-byte protocol field in the IPv4 header, there aren't likely to be any hash collisions.

It is important to differentiate this protocol registration facility from the protocol switch table protosw. The protocol switch table is for socket layer de-multiplexing. There is a separate set of interface functions associated with the protocol switch table, and they are discussed in Chapter 5.

6.7.3 The Transport Protocol Dispatch Process

Conceptually, the dispatch of the transport layer protocol should be simple, IP gets the protocol field value, calculates the hash, indexes into inet_protos, obtains the inet_protocol instance, and dispatches the transport layer protocol's receive function through the handler field. This is what ultimately happens, but it is actually more complicated.

IPv4 receive processing is covered in more detail in Chapter 9. However, in this chapter, we discuss part that involves the inet_protos array. Hand-off from IP to the transport layer uses the packet routing facility in IP, and this the same mechanism used for external routing. Internal routing involves the destination cache (see Section 6.9), the routing table, rtable, and the forwarding information base, fib_table. Packet routing is discussed in Chapter 9.

We begin by looking at the IPv4 receive routine. The receive routine, ip_rcv is passed a packet in a socket buffer in the parameter skb.

```
int ip_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type
*pt);
```

We check the dst field in skb that points to the destination cache entry to differentiate between packets we are supposed to transmit and packets that must be internally routed. If dst is not NULL, it means that that IP already "knows" where to send this packet. If dst is NULL, we must find a route, either internal or external, for the packet. To do this, the skb is passed to the IP router to determine the route and whether it is an external route or an internal destination. Ip_route_input fills in the destination field, dst in the skb. As part of the process, the IP router checks the type field in the routing result structure, fib_result.

```
struct fib_result
{
    unsigned char    prefixlen;
    unsigned char    nh_sel;
    unsigned char    type;
    unsigned char    scope;
    struct fib_info  *fi;
#ifdef CONFIG_IP_MULTIPLE_TABLES
    struct fib_rule   *r;
#endif
} ;
```

If type is set to RTN_LOCAL, the packet is sent to the transport layer protocol by setting the destination cache's input field to ip_local_deliver. In addition, if the entry in the routing table entry has the flag RTCF_LOCAL set, we know we must route this packet locally. Since the ip_local_deliver is set in the destination cache entry, it will be executed when the packet is sent. Later, when ip_local_deliver is called, it calculates a hash value from the protocol field in the IP header. This hash value is used to find an entry in the inet_protos table to select the transport protocol that is the destination for this packet. Finally, the destination transport protocol's input function is called through the handler field of the entry in inet_protos.

6.8 Cache Rich

Linux contains several important cache facilities used with routing and address translation. On of these caches is the routing table cache and contains the cached entries for the recently used routes. The other cache is the neighbor cache, in IPv4 commonly called the ARP cache. The ARP cache contains the mapping of the IP address at Layer 3 with the link layer address used at Layer 2.

Routing at OSI Layer 3 and address translation at OSI Layer 2 are conceptually very different. Layer 3 is the network layer. Its cache is really a front end for a complete routing table used to facilitate decisions about which next-hop machine to send packets. The Layer 2 cache is for address translation and essentially is a lookup table where IP addresses can be mapped into hardware addresses. However, if a route in the table points to a reachable destination, there probably already is an ARP cache entry for the same destination. In an optimized system, we want to avoid multiple table lookups during processing of packets, therefore, we have the route

entry reference the ARP cache entry. In any real-world protocol implementation such as Linux TCP/IP, these two functions are closely related. There is a benefit for some of the underlying data structures to be shared.

Both of the route cache and the ARP facilities generally cache entries for performance reasons, because a cache provides for fast lookup without having to traverse many entries. In addition, they both require timeouts that are associated with each of the table entries. Linux implementation stresses performance, so IP uses some shared infrastructure to implement both caches. As the reader will know from Chapter 3, ARP is the protocol commonly used for address translation in IPv4. In some other TCP/IP stack implementations, ARP is implemented with a separate table in memory called the ARP cache. Each entry in the ARP cache contains a destination IP address and a corresponding physical address, and the destination physical address is inserted into the destination field of the MAC header when a packet is transmitted. All TCP/IP implementations supporting Ethernet must implement the ARP protocol. Earlier implementations tended to hard code their implementations for ARP and Ethernet and don't provide any type of general address translation facility. In contrast, Linux has a flexible mechanism to do address translation independent of Ethernet or any other specific MAC addressing scheme and independent of the ARP protocol. An advantage of using a generic mechanism is that the same infrastructure can be used for IPv4, IPv6, ARP, and other protocols.

There are actually three interacting mechanisms involved in network address translation. The first is the *destination cache*. It is a generic holder for destination addresses and functions in a form independent of the specific protocol. The second mechanism, the *hardware header cache* (HH), holds the MAC layer header and pointers to functions to manipulate the header. The third cache is called the *neighbor cache*. It maintains information about all locally reachable nodes on the network.

The neighbor cache is used primarily by neighbor discovery in IPv6 [RFC 2461], but is also used by ARP in IPv4 and is available for use by other address resolution protocols besides ARP.

These three caches are part of a complex mechanism that might seem hard to sort out at first. Together, the three caches do most of the work of address translation. They provide a framework for any protocol to maintain its address translation table, and allow device drivers to easily access the link layer header information. It is a very flexible system, but might seem to be a complicated way to do a simple job. Extensive use of hashing techniques makes address translation quite efficient so packet processing speed is not impacted.

The detailed discussion of IP routing is in Chapter 9. However, in this chapter we will show important fields in the data structures and explain the primary functionality for address translation. Next, we will walk through the transmission sequence to show how the data structures are used to do address translation.

6.8.1 Destination Cache

The destination cache entry contains all the information for a resolved destination. The destination can be an internal destination for incoming packets or can resolve to a specific hardware address for an external machine. It is through this structure that the link layer header is

obtained for transmitted packets. This facility is generic and protocol independent; it is not just used by IP.

The destination cache is really a library of generic functions and structures that serves as part of the framework for Linux. Specific implementations of the destination cache can be found in the ARP protocol for IPv4 and the ND protocol for IPv6.

The dst_entry structure defines an individual entry in the destination cache.

```
struct dst_entry
{
```

Most of the important fields are explained except those related to the internal hash processing. The first field, next, is a pointer to next dst_entry on the list. A routing table cache entry (described in Section 9.3.1 in Chapter 9) can also point to a dst_entry structure, so either type of structure can be treated identically.

```
    struct dst_entry    *next;
```

The following field contains the reference count for the entry in the destination cache __refcnt. When the destination cache entry is freed, the reference count is decremented. A zero value means garbage collection can be performed on this entry. A nonzero value keeps the cache entry from being deleted if it is still being referenced. The destination cache entry structure must be locked before changing these two fields. The next field, __use, is the use count. It is incremented when a route is being used.

```
    atomic_t            __refcnt;
    int                 __use;
    struct dst_entry    *child;
```

Dev points to either an input network interface device or an output network interface device depending on whether this destination is internal or external. The next field, obsolete, indicates that this entry is no longer used. If the value is greater than one, it means that the entry has been returned to the slab cache.

```
    struct net_device   *dev;
    int                 obsolete;
```

Flags is set to the flags for this destination entry.

```
    int                 flags;
```

Flags can contain the following four values.

```
#define DST_HOST        1
#define DST_NOXFRM      2
#define DST_NOPOLICY    4
#define DST_NOHASH      8
```

Lastuse is set to the time in ticks that this entry was allocated. It indicates the age of cache entry. Expires is the time when this cache entry ages out. It is used by the garbage collection facility to determine which entries are obsolete. For example, when these entries are used for routing cache, expires indicates which routes have aged out.

```
    unsigned long       lastuse;
    unsigned long       expires;
```

The next two fields are for rate limiting of the ICMP protocol.

```
    unsigned long       rate_last;
    unsigned long       rate_tokens;
    int                 error;
```

The next field, neighbour, is a pointer to neighbor cache entry that is used for address resolution. For more information about the neighbor cache, see Section 6.8.4. If not NULL, hh points to the hardware header cache for this destination. If the destination cache entry is for a packet with an internal destination, hh will be NULL. For output packets, hh will be used to get the link layer header.

```
    struct neighbour    *neighbour;
    struct hh_cache     *hh;
```

The next field, xfrm_state, points to a transformation instance for the Security Policy Database (SPD).

```
    struct xfrm_state   *xfrm;
```

The next two fields, input and output, point to functions that are called when a packet is processed using a particular destination cache entry. Input is the default input function. When dst_entry is first created, this field is initialized to dst_discard. For example, for cache destinations used in routing input packets intended for local consumption, this field points to the function ip_local_deliver. Output is the default output function. This function is called by the protocol's output function if hh and neighbour are NULL. When a destination cache entry is created, output is initialized to dst_blackhole. For example, when destination cache entries indicate that IPv4 output packets should be routed, output will point to ip_forward.

```
    int                 (*input)(struct sk_buff*);
    int                 (*output)(struct sk_buff*);
```

Tclassid contains traffic classes and is used for traffic shaping. It is defined if the class-based routing option is configured into the Linux kernel.

```
#ifdef CONFIG_NET_CLS_ROUTE
    __u32                tclassid;
#endif
```

The next field, ops, is the set of operation functions for the destination cache. This structure is initialized to a set of functions specific to a network layer protocol. They are usually statistically

defined when the network layer protocol is initialized. For example, IPv4 initializes dst_ops to ipv4_dst_ops in file *linux/net/ipv4/route.c*.

```
    struct  dst_ops         *ops;
    struct rcu_head         rcu_head;
    char                    info[0];
} ;
```

Although it has an initialization function, the destination cache is instantiated when it is used as part of ARP or ND. The initialization function, dst_init, is defined in file *linux/net/core/dst.c*.

```
void __init dst_init(void)
{
```

The only thing we do in this function is register the destination cache with the network device notifier so destination cache entries will be automatically removed when a network interface device goes down or is disconnected.

```
    register_netdevice_notifier(&dst_dev_notifier);
}
```

The notifier block, dst_dev_notifier, consists of only one function, dst_dev_event, which is discussed in Section 6.8.3.

```
struct notifier_block dst_dev_notifier = {
    dst_dev_event,
    NULL,
    0
} ;
```

6.8.2 The Destination Operation Structure

The destination operation structure contains pointers to operation functions for the destination cache entry. It is largely through this structure that the destination cache takes on its personality as the ARP cache or neighbor discovery cache. The dst_ops structure is accessed through the ops field in the dst_entry described previously.

```
struct dst_ops
{
```

The first field, family, is the address family for this cache entry, such as AF_INET. The next field, protocol, is the same as the link layer header protocol field value. For example, if this entry were for IP, protocol would be ETH_P_IP.

```
    unsigned short          family;
    unsigned short          protocol;
```

Gc_thresh is the garbage collection threshold value. If the destination cache entry is doubling as a routing cache entry, this field is equal to the size of the routing cache hash table plus one.

```
    unsigned                gc_thresh;
```

Gc is the garbage collection function for this entry, and check is a pointer to the function to check the validity of this cache entry.

```
    int                     (*gc)(void);
    struct dst_entry        *(*check)(struct dst_entry *, __u32 cookie);
```

The next field, destroy points to the destructor function for this entry.

```
    void                    (*destroy)(struct dst_entry *);
```

The next field is the negative advice function pointer, negative_advice, which is called when we receive a redirect ICMP packet that negates this entry in the destination cache. Link_failure is a pointer to a function that says that a previously good entry is no longer valid because of a dropped link.

```
    struct dst_entry        *(*negative_advice)(struct dst_entry *);
    void                    (*link_failure)(struct sk_buff *);
    void                    (*update_pmtu)(struct dst_entry *dst, u32 mtu);
    int                     (*get_mss)(struct dst_entry *dst, u32 mtu);
```

Entry_size is the size of the destination cache entry. Entries contains the number of entries in a particular destination cache implementation and is atomically incremented and decremented. Kmem_cachep is the pointer to the slab cache from which destination cache entries are allocated.

```
    int                      entry_size;
    atomic_t                 entries;
    kmem_cache_t             *kmem_cachep;
} ;
```

6.8.3 Destination Cache Utility Functions

There are a few important utility functions associated with the destination cache. Several of these functions are called directly by users of the destination cache, and some of them are generic functions for manipulating entries that are initialized in the destination cache entry when it is created. The functions are defined in file *linux/net/core/dst.c*. Destination cache entries are allocated with dst_alloc.

```
void * dst_alloc(struct dst_ops * ops)
{
```

In this function, we allocate an instance of a dst_entry structure. In most cases, the callers will have initialized an instance of a dst_ops structure at compile time. Then, we are called with a pointer to the dst_ops instance each time a new destination cache entry is requested.

```
    struct dst_entry * dst;
```

We check that entries in dst_ops is greater than the garbage collection threshold, gc_thresh. By checking the values in the destination operations, in effect we are checking the values specific to the destination cache user.

```
    if (ops->gc && atomic_read(&ops->entries) > ops->gc_thresh) {
```

If the threshold is exceeded, it calls the cache garbage collection function through the gc field of dst_ops.

```
        if (ops->gc())
            return NULL;
    }
```

We allocate the entry from the slab cache pointed to by kmem_cachep in the dst_ops structure.

```
    dst = kmem_cache_alloc(ops->kmem_cachep, SLAB_ATOMIC);
    if (!dst)
        return NULL;
    memset(dst, 0, ops->entry_size);
    atomic_set(&dst->__refcnt, 0);
```

Then it sets the ops field to the dst_ops pointer passed in as a parameter. It sets lastuse to the current time, jiffies, and atomically increments entries. It sets input to dst_discard and output to dst_blackhole.

```
    dst->ops = ops;
    dst->lastuse = jiffies;
    dst->path = dst;
    dst->input = dst_discard;
    dst->output = dst_blackhole;
#if RT_CACHE_DEBUG >= 2
    atomic_inc(&dst_total);
#endif
    atomic_inc(&ops->entries);
    return dst;
}
```

Destination cache entries are freed by calling dst_free, an inline function defined in file *dst.h*.

```
static inline void dst_free(struct dst_entry * dst);
```

First, dst_free locks the entry. If __refcnt is zero, the entry is immediately destroyed. If not, it re-initializes dst_entry and places it on the garbage collection list, dst_garbage_list, to be destroyed when the garbage collection timer executes.

Dst_destroy immediately removes a cache entry.

```
void dst_destroy(struct dst_entry * dst);
```

The first thing we do in dst_destroy is check to see if there is an attached neighbor cache entry. If so, we free the neighbor. Then, we check for a hardware cache entry, and if there is one, we free it. Before calling kmem_cache_free to return dst to the slab cache, we call the higher-level destroy function by de-referencing the destroy function pointer referenced in the ops field of dst_entry.

The next two functions, dst_clone and dst_hold, both atomically increment the reference count in the __refcnt field. Both of these functions check that the argument dst is not NULL, but dst_hold assumes that the argument is valid.

```
static inline struct dst_entry * dst_clone(struct dst_entry * dst);
static inline void dst_hold(struct dst_entry * dst);
```

Dst_confirm calls neigh_confirm, passing it the value in the neighbour field. Neighbor confirmation verifies that the neighbor entry is reachable.

```
static inline void dst_confirm(struct dst_entry *dst);
```

The two functions, dst_link_failure and dst_negative_advice, call the higher-level negative_advice and link_failure functions through the respective pointers in the ops field of dst_entry.

```
static inline void dst_link_failure(struct sk_buff *skb);
static inline void dst_negative_advice(struct dst_entry **dst_p);
```

6.8.4 Destination Cache Garbage Collection

In this section, we will discuss how the destination cache Garbage Collection (GC) works. In addition to the default GC, the higher-level users of the destination cache may also have a separate garbage collection capability with a GC function reached through the gc field of the dst_ops structure. This is how the facility for aging out cached routes works and explicit garbage collection of cached routes is covered in Chapter 9. Here we discuss generic low-level destination garbage collection, implemented in file *linux/net/core/dst.c*. In addition, we also show how device status change events are handled for network interface devices.

The state governing garbage collection of dead destination cache entries is protected by a static global lock, dst_lock.

```
static spinlock_t        dst_lock = SPIN_LOCK_UNLOCKED;
```

The destination cache maintains a garbage collection list for stale entries. Entries may be placed on the garbage collection list either in a bottom-half context or by a higher-level kernel facility. However, the garbage collection function itself executes in the bottom-half context by the garbage collection timer. Dst_lock protects entries while they are being added to the garbage collection list. The garbage collection timer is called dst_gc_timer and it is defined in file *linux/net/core/dst.c*.

```
static struct timer_list dst_gc_timer = TIMER_INITIALIZER(dst_run_gc,
0, DST_GC_MIN);
```

DST_GC_MIN is the minimum expiration time for an entry. It is never less than one second because we never would have route cache entries time out that fast. The timer expiration values are altered so the timer execution is somewhat randomized, and so does not self-synchronize and cause a large number of routes to age out at once.

The timer function, dst_run_gc, executes when the garbage collection timer expires.

```
static void dst_run_gc(unsigned long dummy);
```

Dst_run_gc is not a complex function, and most of what we do involves calculating a new timer expiration value. The first thing we do is to try the lock, dst_lock, to see if the GC list is locked. If so, we simply call mod_timer to restart the timer and return. If it survives this test, we delete the existing timer and then walk through the garbage collection list, dst_garbage_list, and call the dst_destroy function pointer in dst_ops for each entry on the list.

We saw earlier that the destination cache registers with the netdevice_notifier when it is initialized. This is because the destination cache will want to post all entries associated with the interface for GC.

The function that will receive device notification events is dst_dev_event, which is also in file *net/core/dst.c*.

```
static int dst_dev_event(struct notifier_block *this, unsigned long
event, void *ptr);
```

The parameter this points to the notifier_block. Event contains the event type we are looking for. We actually are interested in two event types, NETDEV_DOWN and NETDEV_UNREGISTER. The third argument, ptr, is actually the net_device structure pointer back to the device that is telling us about the event. In dst_dev_event, we walk through the garbage collection list to see which entries refer to the device that is sending the event. If we are receiving the NETDEV_DOWN event, we set the input field of the dst_entry to dst_discard and the output field to dst_blackhole. This is to ensure that any transmit or receive packets queued up for the device will be discarded. If we are receiving the NETDEV_UNREGISTER event, we set the device to point to the loopback device, loopback_dev.

6.8.5 The Neighbor System

As discussed earlier, the ARP protocol is implemented using the Linux generic destination cache. In addition, we know that we have a generic destination cache facility to allow the same data structures to be used for the Neighbor Discovery (ND) protocol [RFC 2461] for IPv6. The destination cache is used by the neighbor system to provide a precalculated “fast path” for output packets once a destination address has been resolved. This provides faster packet processing for connected TCP sockets. Since many packets are going to the same destination, we want to be able to quickly build the packet from known information and avoid time-consuming searches of the routing table and ARP cache for each packet. If a destination is connected, the fast path is used and the protocol’s output packet will be internally routed in such a way that the packet’s destination address can be quickly pre-pended to the packet.

A neighbor table instance consists of a list of neighbor cache entries accessed through hash tables. Each neighbor table is specific to an address resolution protocol. In this section, we will discuss the neighbor cache entry, the neighbor parameters, and the neighbor table.

The neighbor system includes a few key data structures. The most funda mental one is the neighbor table, neigh_table, defined in file *linux/include/net/neighbour.h*.

```
struct neigh_table
{
```

Each address resolution protocol such as ARP will create one instance of the neigh_table. The table contains a slab cache from which neighbor cache entries are allocated. It also contains pointers to the individual neighbor cache entries and a hash table for quick access to the entries. The neighbor tables are put on a list and the field next points to the next table instance in the list. A neighbor cache entry can be accessed by multiple tables, so a specific cache entry can be used for both IPv4 and IPv6. Family contains the address family associated with this particular neighbor table instance.

The first field in the neigh_table, next, points to the next neighbor cache entry in the list. Family is the address family for this neighbor cache implementation, and for TCP/IP, it will be set to AF_INET.

```
    struct neigh_table *next;
    int                family;
    int                entry_size;
    int                key_len;
```

The hash field points to a hash calculation function. For example, ARP sets this field to the function arp_hash. Neigh_table includes callback functions, constructor, pconstructor, pdestructor, and proxy_redo. When a neighbor table instance is created, these function pointers are initialized to point to protocol-specific functions. Constructor is for creation of a neighbor cache entry, and pconstructor is for creation of a neighbor proxy entry. In general, the fields beginning with a “p” are used for proxy cache entries. The proxy fields are used for proxy ARP.

```
    __u32              (*hash)(const void *pkey, const struct net_device *);
    int                (*constructor)(struct neighbour *);
    int                (*pconstructor)(struct pneigh_entry *);
    void               (*pdestructor)(struct pneigh_entry *);
    void               (*proxy_redo)(struct sk_buff *skb);
    char               *id;
```

Parms points to the neighbor table parameters, discussed later in this section. Along with each table are a GC task and a garbage collection timer. The neighbor table system uses GC as a generic mechanism for aging out cache entries. The fields beginning with "gc" are used by the garbage collection.

```
    struct neigh_parms  parms;
    int                 gc_interval;
    int                 gc_thresh1;
    int                 gc_thresh2;
    int                 gc_thresh3;
    unsigned long       last_flush;
    struct timer_list   gc_timer;
    struct timer_list   proxy_timer;
    struct sk_buff_head proxy_queue;
```

```
    int                 entries;
    rwlock_t            lock;
    unsigned long       last_rand;
    struct neigh_parms *parms_list;
    kmem_cache_t       *kmem_cachep;
    struct neigh_statistics stats;
```

The next two fields, hash_buckets and phash_buckets, are used with hash functions for quick access to the neighbor and proxy neighbor cache entries for this particular neigh_table instance. It is possible, at least theoretically, for a particular neighbour to be accessed through two different neigh_tables instances.

```
    struct neighbour    *hash_buckets[NEIGH_HASHMASK+1];
    struct pneigh_entry *phash_buckets[PNEIGH_HASHMASK+1];
} ;
```

In addition to the neigh_table, there is a structure that contains parameters for manipulating neighbor table cache entries. These are used with the sysctl (System Control) mechanism, if sysctl is configured into the Linux kernel, so entries in the neighbor table cache can be added or deleted.

Along with the other data structures, the neighbor cache system neighbor parameters structure, neigh_parms, is defined in file *linux/include/net/neighbour.h*.

```
struct neigh_parms
{
```

Each entry in the neighbor cache has an associated neigh_parms accessed through the parms field in the neighbor cache entry. Neigh_parms includes configurations items, reachability information, and timeouts for a neighbor cache entry. The values in neigh_parms are set differently for each address resolution protocol.

```
    struct neigh_parms *next;
    int    (*neigh_setup)(struct neighbour *);
    struct neigh_table *tbl;
    int    entries;
    void   *priv;
    void   *sysctl_table;
    int    base_reachable_time;
    int    retrans_time;
    int    gc_staletime;
    int    reachable_time;
    int    delay_probe_time;
    int    queue_len;
    int    ucast_probes;
    int    app_probes;
    int    mcast_probes;
    int    anycast_delay;
    int    proxy_delay;
    int    proxy_qlen;
    int    locktime;
} ;
```

The neighbor system also provides several functions for creation and removal of neighbor cache entries.

The first of these functions, neigh_parms_alloc, allocates an instance of the neigh_parms structure.

```
struct neigh_parms *neigh_parms_alloc(struct net_device *dev,
struct neigh_table *tbl);
```

In this function, the first thing we do is allocate the structure with kmalloc. There is no slab cache associated with neighbor parameters because there is one for each protocol that neighbor does discover, which is not likely to be very many. Next, we initialize the reachability timer to a random value. Next, neigh_parms_alloc puts the neigh_parms structure on the linked list pointed to by parms in the neigh_table.

The next function, neigh_parms_release, removes the neigh_parms structure from the list in the parms field of neigh_table.

```
void neigh_parms_release(struct neigh_table *tbl, struct neigh_parms
*parms);
```

So far, we have not discussed the neighbor cache entry and the neighbor parameter structure. Now we will look at the neighbor cache entry itself. Each neighbor table consists of a hash table of neighbor cache entries.

Each neighbor cache entry is contained by the neighbour structure.

```
struct neighbour
{
```

Each entry is implemented on a list, and next is the pointer to the next entry. Tbl is a pointer back to the particular neighbor table, neigh_table, which includes this entry.

```
    struct neighbour    *next;
    struct neigh_table  *tbl;
```

Next, parms is a pointer to the neighbor parameters described earlier, and dev points to the network interface for this entry.

```
    struct neigh_parms  *parms;
    struct net_device   *dev;
    unsigned long       used;
    unsigned long       confirmed;
    unsigned long       updated;
    __u8                flags;
```

Nud_state contains the Neighbor Unreachability State (NUD), the values of which are shown in Table 6.4. It is initialized to NUD_NONE when a neighbor cache entry is created.

```
    __u8                nud_state;
```

```
    __u8                type;
    __u8                dead;
    atomic_t            probes;
    rwlock_t            lock;
```

Table 6.4: Neighbor Cache Entry NUD States

State	Value	Description
NUD_INCOMPLETE	0x01	Currently, address resolution is being performed for this neighbor entry.
NUD_REACHABLE	0x02	Indicates that the neighbor is reachable. Positive confirmation was received and the path to this neighbor is okay.
NUD_STALE	0x04	More than the configured elapsed time has passed since reachability confirmation was received for this neighbor.
NUD_DELAY	0x08	More than the configured elapsed time has passed since reachability was confirmed for this neighbor. This state allows TCP to confirm the neighbor. If not, a probe should be sent after the next delay time has elapsed.
NUD_PROBE	0x10	A solicitation has been sent and we are waiting for a response from this neighbor.
NUD_FAILED	0x20	Indicates that neighbor reachability state detection failed.
NUD_NOARP	0x40	Pseudo-state indicating that ARP is not used for this neighbor entry.
NUD_PERMANENT	0x80	Pseudo-state indicating that garbage collection should not be performed to remove this entry.
NUD_NONE	0x00	No state is defined.

The next two fields, ha and hh, contain the hardware header cache used to access the link layer header for an output packet. Hh points to the hardware header cache entry. Hh is not NULL when this neighbor cache entry is resolved to an external destination.

```
    unsigned char       ha[(MAX_ADDR_LEN+sizeof(unsigned long) 1)
&~(sizeof(unsigned long)-1)];
    struct hh_cache     *hh;
```

Refcnt is incremented for each reference to this neighbor cache entry.

```
    atomic_t            refcnt;
```

Output points to the function for transmitting a packet when the hardware header is resolved. It is initialized to neigh_blackhole, which discards the packet when the neighbor cache entry is first created.

```
    int                 (*output)(struct sk_buff *skb);
```

If this neighbor cache entry is for the ARP protocol, arp_queue is the queue of packets waiting for address resolution to be completed. Using a queue is preferable to forcing all packets to wait because it does not tie up the output device while waiting for ARP responses. Ops is a list of protocol-specific functions used for this neighbor cache entry.

```
    struct sk_buff_head arp_queue;
    struct timer_list   timer;
    struct neigh_ops    *ops;
    u8                  primary_key[0];
} ;
```

6.8.6 Neighbor System Initialization

The neighbor system supports multiple instantiation; there are no static data structures. This is because we want each protocol family to have its own separate neighbor table. For example, take IPv4. The AF_INET address family includes ARP, the address resolution protocol [RFC 826]. ARP initializes an instance of the neighbor table when it is initialized. After looking at the generic neighbor table initialization functions, we will look specifically at what ARP does when it creates a neighbor table.

The function that ARP and other resolution protocols call for neighbor table instance creation is neigh_table_init.

```
void neigh_table_init(struct neigh_table *tbl)
{
```

Now is the current time. Neighbor discovery protocols require messages to be sent out at random intervals. To accomplish this, reachable_time in the neigh_parms structure is set to a random value calculated from the seed, base_reachable_time.

```
    unsigned long now = jiffies;

    tbl->parms.reachable_time = neigh_rand_reach_time(tbl >
parms.base_reachable_time);
```

We create a slab cache entry from which neighbor table entries will be created.

```
    if (tbl->kmem_cachep == NULL)
        tbl->kmem_cachep = kmem_cache_create(tbl->id,
                                             (tbl->entry_size+15)&~15,
                                             0, SLAB_HWCACHE_ALIGN,
                                             NULL, NULL);
```

Next, garbage collection is initialized. Neigh_periodic_timer re-calculates the reachability time from a random value.

```
#ifdef CONFIG_SMP
    tasklet_init(&tbl->gc_task, SMP_TIMER_NAME(neigh_periodic_timer),
(unsigned long)tbl);
#endif
    init_timer(&tbl->gc_timer);
```

```
    tbl->lock = RW_LOCK_UNLOCKED;
    tbl->gc_timer.data = (unsigned long)tbl;
    tbl->gc_timer.function = neigh_periodic_timer;
    tbl->gc_timer.expires = now + tbl->gc_interval + tbl->
parms.reachable_time;
    add_timer(&tbl->gc_timer);
```

The proxy timer is used for proxy cache entries.

```
    init_timer(&tbl->proxy_timer);
    tbl->proxy_timer.data = (unsigned long)tbl;
    tbl->proxy_timer.function = neigh_proxy_process;
    skb_queue_head_init(&tbl->proxy_queue);

    tbl->last_flush = now;
    tbl->last_rand = now + tbl->parms.reachable_time*20;
    write_lock(&neigh_tbl_lock);
```

This neighbor table instance is placed on a linked list of neighbor tables.

```
    tbl->next = neigh_tables;
    neigh_tables = tbl;
    write_unlock(&neigh_tbl_lock);
}
```

6.8.7 ARP and the Neighbor Table

We will look at how the ARP sets up the neighbor system in the file *linux/net/ipv4/arp.c*. Before calling the neighbor table initialization function, ARP creates the ARP cache by defining an instance of the neigh_table structure.

```
struct neigh_table arp_tbl = {
   .family =       AF_INET,
   .entry_size =   sizeof(struct neighbour) + 4,
   .key_len =      4,
   .hash =         arp_hash,
    .constructor =  arp_constructor,
    .proxy_redo =   parp_redo,
    .id =           "arp_cache",
    .parms = {
        .tbl =      &arp_tbl,
        .base_reachable_time = 30 * HZ,
        .retrans_time =        1 * HZ,
        .gc_staletime =        60 * HZ,
        .reachable_time =      30 * HZ,
        .delay_probe_time =    5 * HZ,
        .queue_len =           3,
        .ucast_probes =        3,
        .mcast_probes =        3,
        .anycast_delay =       1 * HZ,
        .proxy_delay =        (8 * HZ) / 10,
        .proxy_qlen =         64,
        .locktime =           1 * HZ,
    } ,
    .gc_interval =  30 * HZ,
```

```
    .gc_thresh1 =   128,
    .gc_thresh2 =   512,
    .gc_thresh3 =   1024,
} ;
```

The initialization function for ARP is arp_init.

```
void __init arp_init(void)
{
```

We initialize a neighbor table for the ARP cache by calling neigh_table_init.

```
    neigh_table_init(&arp_tbl);
    dev_add_pack(&arp_packet_type);
    arp_proc_init();
```

Here we are registering neigh_parms with the sysctl (System Control) mechanism if sysctl is configured into the Linux kernel.

```
#ifdef CONFIG_SYSCTL
    neigh_sysctl_register(NULL, &arp_tbl.parms, NET_IPV4,
                          NET_IPV4_NEIGH, "ipv4");
#endif
```

Finally, we register with the net_device notifier, so ARP will be told when a network interface has a change in status.

```
    register_netdevice_notifier(&arp_netdev_notifier);
}
```

6.8.8 Neighbor System Utility Functions

Linux provides a group of neighbor cache utility functions to create and remove neighbor cache entries, resolve addresses, and check the reachability state of a neighbor cache entry. These functions are defined in file *linux/net/neighbour.c*.

The first of these functions, neigh_lookup, finds a neighbor cache entry in the neighbor table, tbl, from the key, pkey, and the network interface device, dev.

```
struct neighbour *neigh_lookup(struct neigh_table *tbl, const void
*pkey, struct net_device *dev);
```

Generally, pkey will be an IP address.

The next function, __neigh_lookup, also finds a neighbor cache entry. It is an inline function implemented in *linux/include/net/neighbour.h*.

```
struct neighbour *__neigh_lookup(struct neigh_table *tbl, const void
*pkey, struct net_device *dev, int creat);
```

It returns a pointer to the neighbor cache entry if it finds one. If not, it creates a new neighbor cache entry if the parameter created is nonzero.

The next function, neigh_create, creates a new neighbor cache entry in neigh_table using pkey and dev. As with neigh_lookup, pkey is a lookup key, and for IPv4, it is generally an IP address.

```
struct neighbour * neigh_create(struct neigh_table *tbl, const void
*pkey, struct net_device *dev);
```

This function, Neigh_destroy, deletes all resources used by a neighbor cache entry, neigh.

```
void neigh_destroy(struct neighbour *neigh);
```

It frees all references to the link layer header in the hh field of neighbour and removes all socket buffers from the ARP queue, arp_queue.

The neigh_event_send function is for sending an event based on the entry's NUD state.

```
int  neigh_event_send(struct neighbour *neigh, struct sk_buff *skb);
```

The exact event depends on the protocol. For example, if the neighbor table is an ARP cache, it could be an ICMP message or an ARP *who-has* request depending on the current NUD state.

The next function, neigh_release, releases a neighbor cache entry.

```
void neigh_release(struct neighbour *neigh);
```

We release the entry by atomically decrementing the reference count. If the count is zero, we call neigh_destroy.

The next function, neigh_confirm, updates the neighbor cache entry.

```
void neigh_confirm(struct neighbour *neigh);
```

It marks the entry with the current time.

This function, neigh_is_connected, checks to see if the NUD state is connected.

```
int neigh_is_connected(struct neighbour *neigh);
```

The neigh_clone function clones a neighbor cache entry.

```
struct neighbour * neigh_clone(struct neighbour *neigh);
```

It does not copy the entry; it just increments the reference count.

This function, neigh_update, updates the neighbor cache entry.

```
int  neigh_update(struct neighbour *neigh, const u8
```

```
*lladdr, u8 new, int override, int arp);
```

The parameter override says to update the entry even if the link layer address in lladdr is different from the one in the existing entry. If the parameter arp is set to a nonzero, neigh_update checks to make sure the neighbor cache entry is an ARP entry.

This function, neigh_resolve_output, resolves a neighbor cache entry.

```
int  neigh_resolve_output(struct sk_buff *skb);
```

It is for resolving a destination address, and sending an event if it needs to. The following two functions are both output functions.

```
int  neigh_connected_output(struct sk_buff *skb);

int  neigh_compat_output(struct sk_buff *skb);
```

These two functions, neigh_add and neigh_delete, are for creating and deleting neighbor cache entries in response to user requests. They are called from the netlink internal messaging protocol.

```
int neigh_delete(struct sk_buff *skb, struct nlmsghdr *nlh, void *arg);
int neigh_add(struct sk_buff *skb, struct nlmsghdr *nlh, void *arg);
```

The next function, neigh_dump_info, dumps the neighbor cache table.

```
int neigh_dump_info(struct sk_buff *skb, struct netlink_callback *cb);
                    struct netlink_callback *cb)
```

It calls the function pointed to by cb.

There is also a corresponding set of functions to manipulate the proxy neighbor cache entries. The first two, pneigh_lookup and pneigh_delete, are similar to the nonproxy versions.

```
struct pneigh_entry * pneigh_lookup(struct neigh_table *tbl, const void
*pkey, struct net_device *dev, int creat);

int pneigh_delete(struct neigh_table *tbl, const void *pkey,
struct net_device *dev);
```

This proxy function, pneigh_enqueue, queues the skb on the proxy queue and sets the proxy timer.

```
void pneigh_enqueue(struct neigh_table *tbl, struct neigh_parms
*p, struct sk_buff *skb);
```

We call pneigh_enqueue if we want the packet to wait for a response from the proxy ARP request.

6.8.9 Sending Packets Using the Destination Cache

The next few sections show how the neighbor structure and the destination cache are used by a protocol's output function to quickly find the output network interface, the hardware header, and the output transmission function once an output packet is about to be transmitted. We will begin at the point where the destination cache has been checked by the network layer protocol's transmit function when a packet is ready to be transmitted. The destination cache entry is accessed through the socket buffer. To understand how the destination cache works, it is best to examine a specific scenario, and we will use the IP protocol's output routine as an example.

In most cases, the final processing step is to transmit the packet out the physical interface. IP output determines what should be done with the packet by checking the dst field in the skb to see if the packet "knows" where it is supposed to go. This is actually a check if the destination hardware addresses has been defined yet for this particular packet.

When the IP protocol is ready to transmit a packet, it checks to see if there is a known destination for the packet by checking the dst field in the skb. Next, ip_finish_output2 checks to see if there is an actual hardware address defined for the destination by checking the hh field in the destination cache entry. The hh field is either NULL or points to the hardware header cache entry containing the complete MAC header for the packet.

6.8.10 Sending Packets Using the Neighbor Cache and Hardware Header Cache

At this point, one of two things can happen depending on the value in the hh field of the destination cache. First, we will see what happens if there isn't a resolved route to the packet's destination. If so, hh would be NULL. If hh is NULL, next we check the neighbor cache by de-referencing the neighbour field of the skb and through this pointer, we get the output function for the neighbor cache entry. In our example, the packet is an IP packet and neighbour has been previously initialized to point to one of the ARP protocol output functions, either neigh_resolve_output or neigh_compat_output. However, if the address of the destination has already been resolved and the hardware header cache is up to date, output will point to the function dev_queue_xmit. Depending on the destination, the output field in the neighbor will be set to one of the function pointers in the neigh_ops structure, output, connected_output, hh_output or queue_xmit. For example, if the network interface is an Ethernet device, ops is set to arp_hh_ops during ARP protocol initialization phase. The ops field defines the behavior of a neighbor cache entry. The ops field is shown in the following code snippet. If this neighbor cache entry is for a reachable node, the output field in the neighbor cache entry is set to point to ops->hh_output.

The structure neigh_ops used in the previous example is defined in the file *linux/include/net/neighbour.h*.

```
struct neigh_ops
{
    int             family;
    void            (*destructor)(struct neighbour *);
    void            (*solicit)(struct neighbour *, struct sk_buff*);
```

```
    void            (*error_report)(struct neighbour *, struct sk_buff*);
    int             (*output)(struct sk_buff*);
    int             (*connected_output)(struct sk_buff*);
    int             (*hh_output)(struct sk_buff*);
    int             (*queue_xmit)(struct sk_buff*);
} ;
```

If the next-hop destination is known but its physical address of the neighbor has not been determined, output will point to neigh_resolve_output. The function neigh_resolve_output checks the neighbor cache's entry state to make sure that the neighbor is reachable. The NUD state is in the nud_state field of the neighbor cache. These states are used to see if there is a need to resolve the destination address. The NUD states are used by the neighbor discovery protocol (NDP) [RFC 2461] and are listed in Table 6.4.

Once the address is resolved, the hh field in the neighbor cache will be updated to point to the hardware header cache entry, which contains the actual link layer destination address. The output function in the neighbor cache entry will be updated to point to the same function as the queue_xmit field in the neigh_ops part of the neighbour structure. This is because neigh_ops had been previously initialized by ARP to point to the dev_queue_xmit function. In any case, dev_queue_xmit is the function that is called to output packet if the address has already been resolved. If the destination is unreachable, the packet will be dropped. Figure 6.3 shows the sequence for transmitting packets through the destination cache.

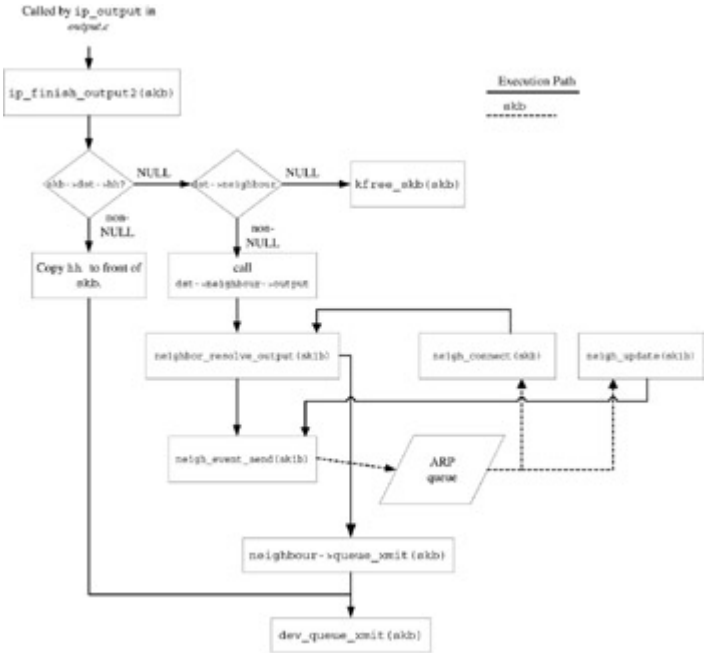

Figure 6.3: Destination cache transmit sequence.

If the hh field is not NULL, IP_finish_output2 gets the link layer header and its length through hh. It reserves space at the head of the sk_buff for the link layer header and copies the header to the front of the packet. The packet is now ready for transmission. Therefore, we call the function pointed to by hh_output and in most cases, it has been initialized to point to the dev_queue_xmit function.

6.9 In_device Structure for IPv4 Address Assignment, Multicast, and Configuration

Any TCP/IP implementation has configurable options for IP multicasting and multiple network interface address assignment. The structure also contains most of the IPv4 tunable parameters. As shown in Chapter 4, most of the net_device structure is protocol independent. However, network interface addresses for IPv4 must be kept somewhere, and they are stored in the in_device structure. Fields can be set in this structure either via sysctrl or setsockopt and the rtnetlink. The address information is kept here rather than the net_device structure because address and Internet configuration information must be kept independent of the device.

The in_device structure is defined in file *linux/include/linux/inetdevice.h*. It is accessed by using the in_dev_get inline function in the same file.

```
struct in_device * in_dev_get(const struct net_device *dev)
```

The in_device structure is shown here.

```
struct in_device
{
```

Dev is a pointer back to the net_device structure that references this Internet device instance.

```
    struct net_device   *dev;
    atomic_t            refcnt;
    rwlock_t            lock;
    int                 dead;
```

This is where we put IP addresses assigned to the interface. Since multiple addresses are supported, ifa_list is implemented as a linked list. Mc_list holds the list of multicast addresses for the device, dev. These multicast addresses are managed by the IGMP protocol covered in Chapter 9, Section 9.8. Mr_v1_seen is used by the IGMP protocol to indicate an incoming IGMP packet.

```
    struct in_ifaddr    *ifa_list;
    struct ip_mc_list   *mc_list;
    rwlock_t            mc_lock;
    struct ip_mc_list   *mc_tomb;
    unsigned long       mr_v1_seen;
    unsigned long       mr_v2_seen;
    unsigned long       mr_maxdelay;
    unsigned char       mr_qrv;
    unsigned char       mr_gq_running;
    unsigned char       mr_ifc_count;
```

The next field is the general query timer.

```
    struct timer_list   mr_gq_timer;
```

This field, mr_ifc_timer, is the interface change timer.

```
    struct timer_list   mr_ifc_timer;
    struct neigh_parms  *arp_parms;
    struct ipv4_devconf cnf;
} ;
```

The ipv4_devconf structure is defined in file *linux/include/linux/inetdevice.h*. It is contained in the cnf field of the in_device structure, shown earlier. It contains configuration values for IPv4, and each field in the structure corresponds to one of the configuration items listed in Table 6.5.

```
struct ipv4_devconf
{
    int    accept_redirects;
    int    send_redirects;
    int    secure_redirects;
    int    shared_media;
    int    accept_source_route;
    int    rp_filter;
    int    proxy_arp;
    int    bootp_relay;
    int    log_martians;
    int    forwarding;
    int    mc_forwarding;
    int    tag;
    int    arp_filter;
    int    medium_id;
    int    no_xfrm;
    int    no_policy;
    void   *sysctl;
} ;
```

Table 6.5: IPv4 Configuration Items

Name	Value	Purpose
NET_IPV4_CONF_FORWARDING	1	IP forwarding is set for this interface.
NET_IPV4_CONF_MC_FORWARDING	2	Multicast forwarding is set for this interface.
NET_IPV4_CONF_PROXY_ARP	3	Respond to proxy ARP requests.
NET_IPV4_CONF_ACCEPT_REDIRECTS	4	Accept ICMP redirects.
NET_IPV4_CONF_SECURE_REDIRECTS	5	Accept secure redirects.
NET_IPV4_CONF_SEND_REDIRECTS	6	Send ICMP redirects.
NET_IPV4_CONF_SHARED_MEDIA	7	Shared media
NET_IPV4_CONF_RP_FILTER	8	Reverse path filtering. This option is turned on by default if IP forwarding is also enabled.
NET_IPV4_CONF_ACCEPT_SOURCE_ROUTE	9	Used for source routing.
NET_IPV4_CONF_BOOTP_RELAY	10	DHCP or Bootp relay is configured.
NET_IPV4_CONF_LOG_MARTIANS	11	This configuration tells us to log packets with Martian addresses.
NET_IPV4_CONF_TAG	12	Not used in core TCP/IP protocol

Table 6.5: IPv4 Configuration Items

Name	Value	Purpose
		suite.
NET_IPV4_CONF_ARPFILTER	13	Configure ARP filtering. ARP filtering can be used to prevent multiple interfaces from responding to an ARP request.
NET_IPV4_CONF_MEDIUM_ID	14	This configuration item is not used in the core TCP/IP protocol suite.
NET_IPV4_CONF_NOXFRM	15	Bypass route policy checks in Linux transformer (XFRM).
NET_IPV4_CONF_NOPOLICY	16	This configuration item turns on the DST_NOPOLICY flag in the destination cache entry as a default value. It also bypasses XFRM policy checks.

Table 6.5 shows the configuration items defined in file *linux/include/linux/sysctl.h*. These variables are associated with the ipv4_devconf structure shown previously. Each variable can be set via sysctl and some by ioctl. There is also a set of version 2.0 compatibility variables with similar names defined in file *linux/include/linux/sysctl.h*.

Several utility functions associated with the in_device structure are used widely in the IP protocol and elsewhere. The most important of these functions is to select the default destination address from the list of addresses assigned to the device. This function is inet_select_address.

```
u32 inet_select_addr(const struct net_device *dev, u32 dst, int scope);
```

Dev points to the network interface device, dst is the destination address associated with a packet that is lacking a source address. We want to find the appropriate source address to place in a packet that is being sent to the destination address defined by the parameter, dst. The parameter, scope is set to either RT_SCOPE_LINK or RT_SCOPE_UNIVERSE. The complete set of values for scope is shown in Chapter 9.

6.10 Security, Stackable Destination, and XFRM

IPSec has been the security mechanism of choice for many years. The security features for TCP/IP were first specified in the mid-1990s. The security architecture for IP, known collectively as *IPSec*, was revised in the late 1990s [RFC 2401]. It was revised to include a definition of an Authentication Header (AH) [RFC 2402], an Encapsulating Security Payload (ESP) [RFC 2406], and Internet Key Exchange (IKE) for key management [RFC 2409]. This book won't go into the theory of network security, but we will explore some of the internal structures used to support IPSec. It also supports the concept of Security Associations (SAs). SAs can be thought of as secure connections. The management of SAs involve a three-tuple consisting of a Security Parameter Index (SPI), a destination address, and a security protocol,

which can be either ESP or SH. With certain types of SAs, the IP traffic passes through an IP tunnel where an outer IP header specifies the IPSec information and an inner header that is the "real" IP header. Linux supports the PF_KEY key management API version 2 [RFC 2367]. The PF_KEY type address family is defined in file *linux/include/linux/pfkeyv2.h*.

To support these security features, Linux provides a Security Policy Database (SPD), which contains the rules to implement the secure policy. It also provides a Security Association Database (SAD) to manage the secure connections. The destination cache, covered earlier in this chapter, is used to assign a destination to a packet, which could be an external host machine or an internal packet handler. Starting with Linux 2.6, there is a mechanism called the Transformer (XFRM) that transforms destination cache entries based on the security policy. The essentials of the xfrm implementation consist of a policy rule implemented as a structure called xfrm_policy. In addition, it defines a transformer with the xfrm_state structure definition from file *linux/include/net/xfrm.h*. The SPD contains the transform rules that are implemented as a list of xfrm_policy instances, ordered by priority. The transformer allows the dst_entry structures to be accumulated into bundles. In addition, the dst_entry structure has a new field, xfrm_state, which points to a xfrm_state instance.

The SPD is accessed when protocol output routines do a lookup in the SPD by calling xfrm_lookup, defined in file *linux/include/net/dst.h*.

```
int xfrm_lookup(struct dst_entry **dst_p, struct flowi *fl, struct sock
*sk, int flags);
```

Xfrm_lookup finds a match in the SPD and then checks the action in the xfrm_policy entry. If the action is XFRM_POLICY_BLOCK, it returns an error. If the action is XFRM_POLICY_ALLOW, then a "bundle" of destinations is accumulated and returned. The xfrm_state structure is defined in file *linux/include/net/xfrm.h*.

There must also be a way for applications to manage the SPD. Linux provides two interfaces between the SPD and the socket layer. The first interface uses PF_KEY type sockets. In addition, messages can be sent to the SPD from the applications using netlink(7). The netlink facility is discussed in Chapter 5.

6.11 Some Practical Considerations

Most of this chapter focused on the infrastructure of Linux TCP/IP. In this section, we present some hints about debugging Linux kernel code and how to access information about each of the kernel modules associated with the protocols that are members of the Linux TCP/IP suite.

6.11.1 Configuring TCP/IP via sysctl and the /proc Filesystem

The /proc filesystem is a virtual filesystem that contains Linux kernel and networking parameters and statistics. The sysctl facility requires the /proc file system to be configured in the kernel. The /proc file system can be browsed by using the usual filesystem navigation commands such as cd and ls. Through the course of the book, we list the /proc facilities that are relevant to the TCP/IP facility being discussed. Elsewhere in this book, we also show examples of how a protocol can

provide support for /proc by registering a function that is called while the user browses the virtual filesystem.

Sysctl parameters can be read or written by a user with the command sysctl(8). In addition to specifying them dynamically, configuration values can be preloaded into the kernel by putting them in the sysctl.conf file, usually in */etc*. See sysctl.conf(8). The best documentation of the sysctl values is in the file *linux/Documentation/networking/ip-sysctls.txt*.

6.11.2 Rate Limiting

One type of configuration parameter that deserves special mention is rate limiting. Rate limiting restricts the timing of various types of packets to provide more predictable real-time performance. This way, rate limiting prevents packets from building up in queues in the device driver or causing system performance bottlenecks elsewhere in TCP/IP. As we have seen in this chapter, the normal Linux clock rate is measured in Hz, the number of ticks per second, and the current time is maintained in the global variable jiffies. The rate limit is set to the allowed number of ticks or jiffies per packet. Some of the rate limits have defaults. For example, our host has a clock rate of 100 ticks per second, which is typical for x86 systems. The default ICMP rate limits are set to 100, allowing one packet per second for certain types of ICMP packets. This is how we could increase the limit of the number of ICMP echo reply packets to the value of two per second. Since the rate limit for ICMP applies to all types of ICMP packet, we must set the ICMP rate mask to add the additional type. We set the value one into icmp_ratemask via the sysctl(8) utility to indicate that we want to rate limit echo reply packets. Then, we set icmp_ratelimit to 50.

In various places in the book, as the path of packets through the source code is traced, we show how rate limits are checked for different packet types. Linux provides the ability to rate-limit many aspects of packet flow through the TCP/IP stack. All sysctl values including rate limits are documented in the file *ip-sysctl.txt* found in *linux/Documentation/networking*.

6.12 Summary

In this chapter, we discussed some of the implementation, or the internal structure, of Linux TCP/IP. The difference between a protocol and its implementation was discussed. We covered the internals of the queuing layer, and how packets pass between the network layer protocols and the device drivers. The mechanism for registering protocol's packet handlers with the stack was covered. We looked at the interface functions available for developers of protocols and talked about some of the internal mechanisms to move packets around within the stack. Chapter 7 covers the Linux system for network buffering in detail. Chapters 8 through 11 look at packets as they are processed—we will follow them as they travel through the layers of the TCP/IP stack.

Chapter 7: Socket Buffers and Linux Memory Allocation

Overview

Chapters 1 through 3 covered general network topics. Chapter 4 began the detailed discussion of the internal TCP/IP stack in the Linux Operating System (OS) by discussing the network interface drivers. Chapter 5 continued by discussing material on the socket layer, and Chapter 6 discussed some of the Linux infrastructure. This chapter continues the detailed discussion of the Linux implementation by discussing the memory allocation scheme and Linux network buffers.

In this chapter, we discuss the Linux socket buffer implementation in detail. Linux socket buffers are used to hold network packets, whether they are outgoing packets allocated at the socket interface or are incoming packets allocated by the network interface driver. Like other Unix implementations, Linux places TCP/IP within the kernel and must use a kernel-based memory allocation scheme to hold network packets. In this chapter, we look at the Linux kernel memory allocation scheme called Linux *slab cache*, and how it is used as the basis for Linux socket buffers. Before we undertake the detailed examination of network packet memory allocation in Linux, we introduce other memory allocation schemes used with TCP/IP implementations and contrast them with Linux. We show how the Linux slab cache has inherent advantages over other methods and how it is appropriate for use in embedded systems' TCP/IP applications.

Memory allocation is a key factor in the performance of any TCP/IP stack. Like other TCP/IP stack implementers before them, the developers of Linux were aware of the importance of memory allocation on network performance. Most other TCP/IP implementations have a memory implementation mechanism that is independent from the OS. However, the Linux implementers took a different approach by using the slab cache method, which is used for other internal kernel allocation, and in this chapter, we will see how this method has been adapted for socket buffers. We'll see how the Linux scheme has sufficient or better perform ance than other memory allocation mechanisms used with other TCP/IP stack implementations.

7.1 Introduction

A complete discussion of memory allocation in modern OSs is beyond the scope of this book. Most modern processors provide hardware memory management. Therefore, most operation systems (with the exception of the smallest OSs used in embedded systems) run in processors that provide support for hardware-based virtual memory. Operating systems use memory management hardware to allow user processes to run in protected memory space. Most virtual memory schemes divide physical memory into small fixed-sized units called *pages*. Memory paging support was originally developed for earlier systems with limited physical memory so each page could be loaded separately into memory from disk. A complete examination of paged virtual memory allocation is also beyond the scope of this book, but we mention it because it is important for understanding the allocation scheme behind the socket buffer implementation. What is important to this discussion is that the page is the fundamental allocation unit used with slab allocation in the Linux kernel. For a good examination of memory allocation used in various

flavors of the Unix operation system, see *Unix Internals*, Chapters 12 through 15 [VAHAL96]. In our discussion, because the TCP/IP implementation is in the kernel, where the physical and virtual addresses are identical, we can ignore virtual memory. However, even without considering virtual memory, the Linux memory allocation scheme remains a complex topic. For a complete discussion on slab memory allocation scheme used in the Solaris OS, refer to Section 12.10 in [VAHAL96]. In this chapter, we will discuss how the Linux slab allocator is used with socket buffers.

Network protocols such as TCP/IP receive, transmit, and process data in relatively small independent packets. Moreover, network protocols also have to run continuously with high reliability and they can't risk any downtime. Yet the OSs must frequently allocate memory buffers for packets as they are received by the computer at wire rates. In addition, the OS must release the buffers at an equally high rate when the stack is done processing the packets. Any latencies associated with memory allocation could have an adverse performance impact. In addition, copying of network buffers should be minimized. Because of the unique needs of network protocols, there are three important requirements for any memory allocator that is to be used with TCP/IP.

- Memory should be made available in different sizes to allow for the variable-length packets used in TCP/IP.
- Copying should be minimized while constructing and processing packets. TCP/IP is constantly processing packets, and this processing involves the frequent pre-pending or removal of headers. The buffering scheme must support this without copying any data. It should be done by moving pointers only.
- Packets need to be easily placed on lists of queues, and packets need to be constructed efficiently from multiple buffers. While processing, multiple buffers of data are often gathered into a single logical packet, and packets are frequently placed on queues to wait for processing or transmission. The buffer data structures should support pointers both for queuing of packets and pulling buffers together into a single packet without copying of data.

Therefore, the memory allocation scheme must be based on a reliable but flexible dynamic memory allocation method that can run continuously. The simplest memory allocation method is based on a heap, but heaps are prone to fragmentation. When buffers are rapidly allocated and returned, the memory space can be quickly broken up into small bits so the time spent searching for an available buffer degrades over time.

Other TCP/IP implementations solve the problem of heap fragmentation by using a separate system of fixed-sized buffers for the TCP/IP stack. In contrast, the Linux implementation of TCP/IP uses the slab memory allocation method also used for allocation by other subsystems in the kernel. The next two sections provide background on earlier buffering schemes used in other TCP/IP stack implementations. Section 7.4 covers the Linux slab allocation method and why it is suitable for use by the TCP/IP stack. Section 7.5 covers the internal structure of skbuffs, the shared area, the header cache, and the internal queuing capability, and Section 7.6 covers the skbuff API.

7.2 Heap-Based Memory Allocation

The first memory allocation method we will look at is heap-based memory allocation. We will show why this method is not appropriate for use by TCP/IP or other network protocols. The simplest method of memory allocation is the basic heap. A heap provides varying-sized buffers that are allocated from a logically continuous space. When an application needs a buffer, it calls a function such as malloc, which returns the smallest available buffer that is larger than the requested amount. The buffer is returned to the heap by calling the function free, which puts the buffer back in the heap. If another user wants a larger buffer, malloc has to pass over all the smaller pieces of memory looking for a contiguous hunk of available space that is larger or equal to the requested amount. For example, if a user wants 97 bytes, he calls malloc(97) , which returns a buffer of 100 bytes. When this buffer is returned by calling free, a 100-byte hole is left in the available memory. Although the 100 bytes is put back on the free list by free, it is only available to a subsequent caller that asks for 100 bytes or less.

As discussed in earlier chapters, a typical networking packet such as an Ethernet frame is 1500 bytes or less in length. Commonly used buffer sizes will be 2K or even 512 bytes or smaller for frequently used small packets. Since a real-world system can have packets arriving on a 100 MB Ethernet connection or higher, packets are being processed at least at rates of one packet per millisecond or less. At these rates, a heap-based best-fit memory allocation scheme would fragment very fast if used in a fast networking application. The fragmentation decomposes the available memory into ever smaller holes. It is possible that without some mechanism to coalesce the smaller holes into larger pieces, the protocol could come to a complete stop.

Heaps are often combined with a resource map consisting of a set of address- and size-tuples and a set of policies for answering memory requests. This method is called a *resource map allocator*. The policy may be first fit, best fit, or worst fit. However, even with the use of one of these policies, the memory pool will fragment over time [KNUTH73]. See Figure 7.1 for an illustration of how the fragmentation could occur.

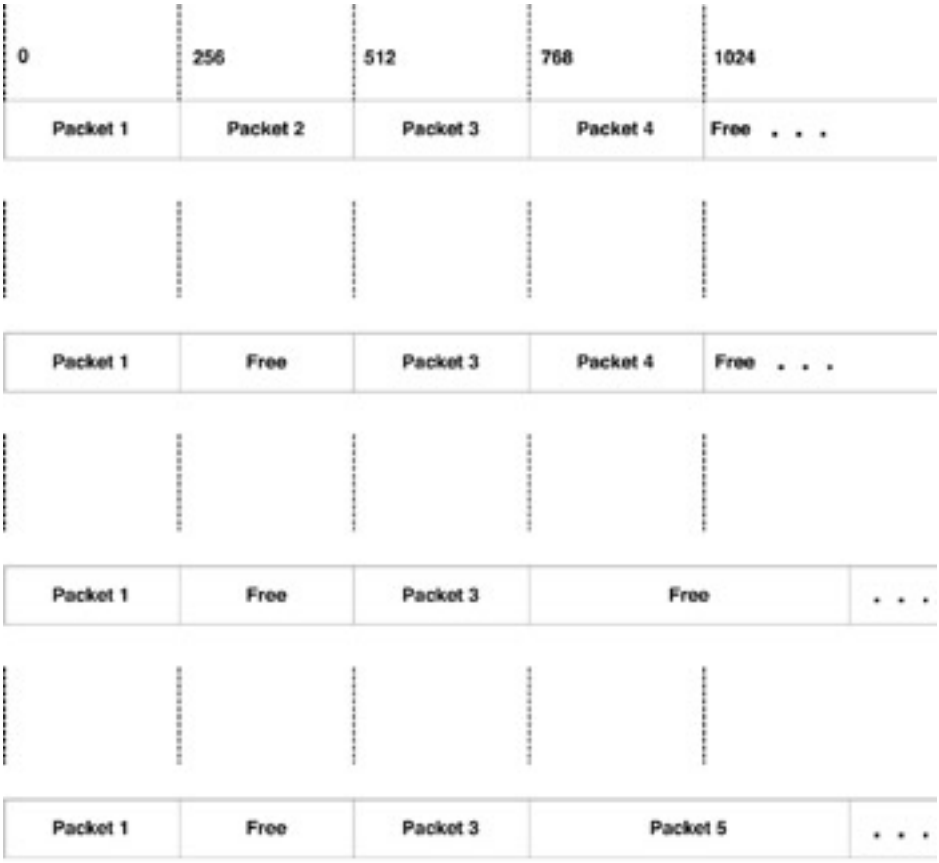

Figure 7.1: Best-fit allocation here.

7.3 Pre-Allocated Fixed-Sized Buffers

Because of the limitation of heap-based methods, an alternative method of memory allocation is often used with networking protocols. This method, based on pre- allocated fixed-sized buffers has better performance and less overhead than a heap-based method. As discussed earlier, a heap-based method of memory allocation is not generally suitable for networking protocols because of its tendency toward fragmentation. In addition, traditionally, in embedded systems it is preferable to have a bounded memory pool. Therefore, most traditional TCP/IP implementations used in embedded systems incorporate a fixed-buffer-based method. The buffer quantities are tuned to allow for the expected amount of networking traffic. Of course, if a buffer becomes available, packets will be dropped until the load drops. An example of the use of fixed-sized buffer allocation can be found in STREAMS [STREAMS93]. STREAMS is a framework for network protocols original developed by Dennis Ritchie and used widely in some Unix variants and other OSs. STREAMS uses a memory allocation method officially called *message buffers*, but more specifically called *mblocks*. In many implementations, this scheme is based on fixed-sized pre- allocated buffers. Buffers are available in sizes that are powers of two. The service routine allocb is called to get a buffer. It will return the next largest-sized buffer, which will hold the requested amount of data. The buffer is taken from the pool of mblocks. Since the buffers are fixed sizes, the buffer pool does not fragment the way the heap-based method does. Refer to Figure 7.2 for an illustration of how packets are allocated with mblocks in STREAMS.

Figure 7.2: STREAMS mblocks.

Another common example of pre-allocated fixed-sized buffers is called mbufs. Mbufs are found in the Berkeley-based TCP/IP implementation that is used in BSD Unix and other proprietary OSs for embedded systems. See [MCKUS96] Section 11.3 for a detailed explanation of mbufs. As is the case with mblocks, mbufs are also intended to support variable-length frames needed for protocol stacks such as TCP/IP. The mbuf data structure is 128 bytes long. It is designed so that the small amounts of data found in short packets or packet headers can be placed in the data area of the mbuf, which is directly below the header. Larger amounts of data are supported by extending the mbufs with clusters.

In many embedded implementations of Berkeley-based TCP/IP, both the mbufs and the clusters are pre-allocated. As with mblocks, clusters are available in sizes that are powers of two. Typical sizes supported can be 64, 128, 256, 512, 1024, and 2048. In most kernels, 2048 is the most common size. Mbufs can also be chained to support queues of packets. Often in embedded systems, network loading factors are predetermined based on the type of network traffic the system is likely to see. Then, parameters are adjusted for the numbers of each size of buffer to be pre-allocated before the system starts up. See Figure 7.3 for an illustration of how the mbuf system works. This presents a challenge for tuners because peak network loads must be anticipated. If the network loading is underestimated, packets can be dropped. Table 7.1 lists the fields in the mbuf structure.

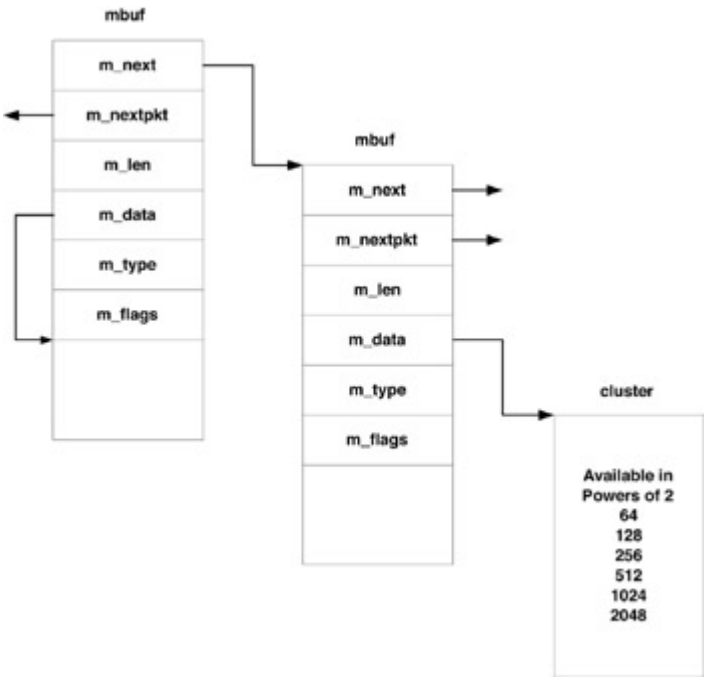

Figure 7.3: Berkeley mbuf structure.

Table 7.1: Mbuf Structure

Field	Purpose
m_next	Next mbuf in chain of mbufs.
m_nextpkt	Next mbuf in a queue of packets.
m_len	This field shows the length of data in the mbuf. It includes the data in the cluster if one is attached.
m_data	Points to the beginning of the usable data. This field can point to the area immediately after the mbuf header, or may point to an address within an attached cluster. This field can be changed to either prepend or remove packet headers.
m_type	Indicates type of mbuf.
m_flags	Indicates whether mbuf is extended with a cluster.
m_pkthdr.len	This location will be the beginning of data if flags field is zero.
m_pkthdr.rcvif	Points to the data structure for interface from which the packet was received.

7.4 Mbufs in BSD 4.4

The BSD 4.4 OS also uses mbufs as the core network buffer system [MCKUS96]. However, there are a few important differences from the fixed-size pre-allocated buffer method described earlier. In BSD 4.4, the mbufs are allocated with the standard kernel memory allocator by calling the malloc() function. Instead of initializing separate lists for each of the cluster sizes, only the list of four-Kbyte clusters is pre-allocated. If smaller cluster sizes are requested they can be

obtained from the free page list of the four-Kbyte clusters. If the list of clusters is exhausted while the protocols are processing packets, the backend page allocator will create more four-Kbyte clusters up to a predefined limit. A reference count is maintained in the clusters. When clusters are de-allocated, they are not returned to the memory pool. Instead, the reference count is decremented and the returned clusters are made available for new cluster allocation.

As with the systems described previously, BSD 4.4 does away with fragmentation. It also eliminates the need for precise pre-tuning of buffer requirements because new buffers are grabbed from the page allocator if needed.

7.5 SLAB Allocation

The Linux operating system uses a memory allocation method called *slab allocation*. This method was first introduced in the Solaris operating system from Sun Microsystems in version 2.4 [VALHAL96]. The slab allocation method is used for all memory allocation in the Linux kernel, not just networking buffers. The slab allocation system is organized into specific slab caches, one for each major function in the Linux kernel. When used for the allocation of network buffers, slab cache allocation has advantages over a heap-based method. The first advantage is that it is less prone to fragmentation. It also has less overhead and uses hardware cache more efficiently than other methods. As discussed earlier, the pre-allocated fixed-sized buffer mechanisms used with most embedded TCP/IP implementations also solve the fragmentation problem. However, a disadvantage of using pre-allocated fixed-sized buffers is that an area of memory has to be pre-allocated for the buffering mechanism and is not available for any purpose other than TCP/IP. Another disadvantage of using the fixed-sized buffering methods is that (except in BSD 4.4) in order to avoid starvation, buffer tuning must be done to determine the amount of buffers of each size, and it is not always possible to predict how an embedded system is going to be deployed.

Slab allocation achieves efficiency by organizing all the memory buffers for a specific purpose into a common cache. The slab cache also makes optimum use of hardware caching, and in Linux, each slab cache can be specifically aligned along a cache line for optimum performance. In Linux, when a memory buffer is de- allocated, it is re-assigned to the cache from which it came instead of going back into a generic pool of buffers. Since each cache is specific to a purpose, common fields can be pre-initialized when the buffer is de-allocated. This is quite efficient because subsequent allocation requests are satisfied from the cache and most of the structure initialization will have already been done. Socket buffers are complex and contain spin locks, use counts, and other complex data structures, all of which require initialization. Network buffers are continually allocated and de-allocated at a high rate as the application writes data into a socket when data is received at the network interface, so pre-initialization of structure fields is particularly beneficial to a networking stack such as TCP/IP. There is considerable gain in having these fields pre-initialized when a buffer is allocated. We will see how Linux benefits by this when we discuss socket buffers later in this chapter.

7.6 Linux SLAB Allocator

The slab allocator can be thought of as having a front end and a back end. The front end consists of the functions to allocate and free cache objects, and the back end is the page memory allocator that assigns one additional slab to a slab cache if there are no objects left in the slab cache. This is called "growing" the slab cache and actually involves assigning one or more memory mapped pages of the right type for the particular slab.

Table 7.2 shows most of the slab caches used in the TCP/IP protocol suite. The first and most important is the socket buffer cache, which is discussed in detail later in this chapter. The other caches are discussed elsewhere in the book as part of the explanation of the various protocols in TCP/IP. The names in Table 7.2 show up as identifiers in the /proc filesystem under /proc/slabinfo.

Table 7.2: Linux TCP/IP Slabs

File Where the Name	Purpose	Cache is Allocated
skbuff_head_cache	Socket buffer cache	core/skbuff.c
flow	Generic flow control cache	core/flow.c
Table ID	Neighbor table cache	core/neighbour.c
sock	Sock structure cache	core/sock.c
sock_inode_cache	Socket inode cache	net/socket.c
udp_sock	UDP socket cache for IPv4	ipv4/af_inet.c
tcp_sock	TCP socket cache for IPv4	ipv4/af_inet.c
raw4_sock	Raw socket cache for IPv4	ipv4/af_inet.c
ip_fib_hash	FIB hash table cache for IPv4	ipv4/fib_hash.c
inet_peer_cache	Internet peer structure cache	ipv4/inetpeer.c
ip_dst_cache	IPv4 routing table destination cache	ipv4/route.c
tcp_open_request	TCP open request cache	ipv4/tcp.c
tcp_bind_bucket	TCP bind bucket cache	ipv4/tcp.c
tcp_tw_bucket	TCP TIME-WAIT state buckets	ipv4/tcp.c
tcp6_sock	TCP socket cache for IPv6	ipv6/af_inet6.c
udp6_sock	UDP socket cache for IPv6	ipv6/af_inet6.c
raw6_sock	Raw socket cache for IPv6	ipv6/af_inet6.c
fib6_nodes	FIB table cache for IPv6	ipv6/ip6_fib.c
ip6_dst_cache	IPv6 destination cache	ipv6/route.c

7.6.1 Linux Slab Cache Utility Functions

Linux provides a set of generic functions to create a slab cache and manipulate the entries in the cache. These functions are all defined in file *linux/include/linux/slab.h*. As discussed earlier, the

slab allocator is used by many kernel subsystems, not just TCP/IP. Therefore, these functions are not part of TCP/IP itself, but are widely used in the sources discussed in this book. In addition, they are available for use by any programmer who is implementing her own protocol module in the Linux kernel.

The function kmem_cache_create creates a new slab cache.

```
kmem_cache_t      *kmem_cache_create (const char *name,
                  size_t size, size_t offset, unsigned long flags,
                  void (*ctor)(void*, kmem_cache_t *, unsigned long),
                  void (*dtor)(void*, kmem_cache_t *, unsigned long));
```

The parameter name is a string pointer that points to the cache's identifier. Generally, it is a hard-coded string such as shown in Table 7.2. The function returns a pointer to kmem_cache_t, which is generically defined as a pointer to the cache structure kmem_cache_s in file *slab.c*. However, each call to kmem_cache_create generally redefines kmem_cache_t to be set to the unique type of the slab cache, which is also the cache name. Flags is set to one of the values in Table 7.3, which are from the file *linux/include/linux/slab.h*. Ctor and dtor point to the constructor and destructor functions for the cache types. They can be NULL, but may point to functions implemented by the caller specific to the cache type.

Table 7.3: Slab Cache Flags

Flag	Value	Purpose
SLAB_DEBUG_FREE	0x00000100UL	Perform debug checks when calling free. This flag does not appear to be used anywhere.
SLAB_DEBUG_INITIAL	0x00000200UL	Perform debug checks when objects are initialized and the constructor is called.
SLAB_RED_ZONE	0x00000400UL	Create a "red zone" around the allocated buffer for overrun checking.
SLAB_POISON	0x00000800UL	Request that a new buffer be initialized with a test pattern, which is a5a5a5a5.
SLAB_NO_REAP	0x00001000UL	Objects in the cache should never be reaped. This flag does not appear to be used very widely.
SLAB_HWCACHE_ALIGN	0x00002000UL	Specifies that objects in the cache be aligned to a hardware cache line when created. This flag is used by almost all the slab caches in TCP/IP.
SLAB_CACHE_DMA	0x00004000UL	Objects in the cache should be created from DMA memory, type GFP_DMA. This flag is not used in TCP/IP. DMA is used by the network drivers, but data is placed into pages attached to the socket buffer and not directly into the slab

Table 7.3: Slab Cache Flags

Flag	Value	Purpose
		cache.
SLAB_MUST_HWCACHE_ALIGN	0x00008000UL	Force alignment to hardware cache line. This flag is not used by TCP/IP.
SLAB_STORE_USER	0x00010000UL	Store the last owner of the object. This flag is used for debugging only.
SLAB_RECLAIM_ACCOUNT	0x00020000UL	Track individual pages to indicate which of them may be reclaimed later. This flag is used by the socket inode cache.
		The following three flags are passed to the constructor function when a cache object is allocated.
SLAB_CTOR_CONSTRUCTOR	0x001UL	Indicates that the constructor should be called for object initialization and not the de-constructor, which normally does object initialization.
SLAB_CTOR_ATOMIC	0x002UL	Indicates to the constructor that it must execute atomically.
SLAB_CTOR_VERIFY	0x004UL	If this flag is set, the constructor is being called to verify only.

The constructor function is called only when new uninitialized slabs are added to the slab cache.

The function kmem_cache_destroy deletes a slab cache.

```
int kmem_cache_destroy (kmem_cache_t * cachep);
```

The parameter cachep points to the slab cache. This function will remove all traces of the slab cache. If a module creates a slab cache that does not need to persist between module loads, this function should be called from the module before it is unloaded. All objects in the cache should be freed before calling this function. The cache must be protected by the caller from allocations while the function is executing.

The next function, kmem_cache_shrink, reduces the size of the slab cache.

```
int kmem_cache_shrink(kmem_cache_t *cachep);
```

It removes as many slabs as possible. It returns a zero when all slabs have been released.

The function kmem_cache_size gets the cache size.

```
unsigned int kmem_cache_size(kmem_cache_t *cachep);
```

It returns the size of the objects in the cache pointed to by cachep.

The next two functions are to allocate and free objects in a particular slab cache. Kmem_cache_alloc allocates an object from a particular slab cache.

```
void * kmem_cache_alloc (kmem_cache_t *cachep, int flags);
```

It returns a pointer to an object allocated from the slab cache pointed to by cachep. The parameter, flags can have one of the four values listed in Table 7.4. Flags is used only when the cache does not contain any objects and it must be "grown." The values are the "get free pages" values, which govern the individual page allocation that occurs in the back end of the slab cache.

Table 7.4: Values for Flag Argument in Kmem_cache_alloc Function

Flag	**Meaning**
GFP_USER	Used when memory is allocated for an application in user space.
GFP_KERNEL	Used with normal kernel memory allocation.
GFP_ATOMIC	Used when the caller is an interrupt service routine.
GFP_DMA	States that memory should be allocated from DMA space. This flag is architecture dependent.

The last function in this section, kmem_cache_free, de-allocates an object from a slab cache.

```
void kmem_cache_free (kmem_cache_t *cachep, void *objp);
```

The parameter objp points to the object to be de-allocated. Cachep must point to the cache from which the object was originally allocated.

7.7 Linux Socket Buffers

In this section, we discuss the internal architecture of the socket buffers, or sk_buffs, how they are allocated, how they are placed in a header cache when freed, and how the internal sk_buff queuing works. Socket buffers are complex and contain many fields with seemingly obscure purposes such as pointers, counters, and other maintenance items. In this and subsequent sections we try to sort out all this data and make sense of all the fields in the sk_buff structure and the functions to manipulate them. The actual data associated with the socket is in a separate buffer, which is generally allocated directly with a call to kmalloc; however, this is not as simple as it seems. A socket buffer can be either cloned or shared, and the socket buffer structure contains fields that indicate whether the buffer is cloned or shared. A utility function is provided to clone a socket buffer, and this function is explained along with the other utility functions later in this chapter. The fields of the socket buffer structure are shown later in this section. The buffer or data area attached to the socket buffer can be in one of various forms. It may directly contain the whole TCP/IP packet, or it may contain only the TCP/IP header with the rest of the packet content in a list of mapped pages. Figure 7.4 shows a picture of a socket buffer in its most general form and how the pointer fields are used to find various positions in the attached TCP/IP packet.

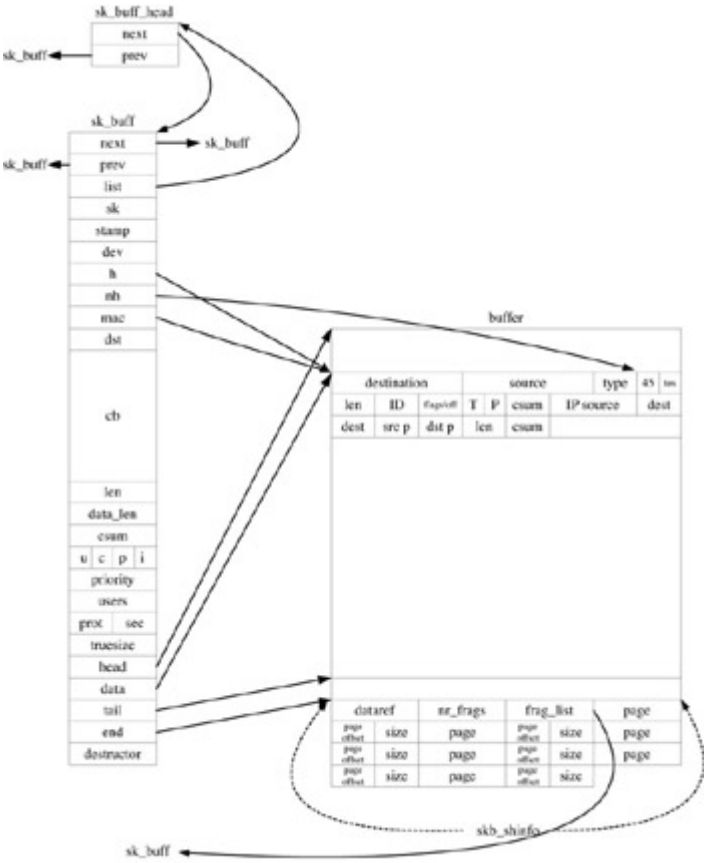

Figure 7.4: Linux socket buffers.

7.7.1 The Socket Buffer Structure, Sk_buff

The socket buffer structure, sk_buff, is defined in file *linux/include/linux/skbuff.h*.

```
struct sk_buff {
```

The following two fields in the socket buffer must be first. As we will see later in the book, sometimes the socket buffer lists are overloaded as different types. Next points to the next buffer on the list, and prev points to the previous buffer.

```
    struct sk_buff      *next;
    struct sk_buff      *prev;
```

The next field, list, points to the head of the list of socket buffers.

```
    struct sk_buff_head*list;
```

The next field points to a sock structure, and for transmitted packets, it points to the socket from which this packet originated.

```
s    struct sock         *sk;
```

For received packets, the next field, stamp, indicates the time this packet arrived at the interface. For received packets, dev shows the device from which this packet was received. Dev points to the network interface. In the case where a pseudo-device is involved, real_dev points to the actual physical device.

```
    struct timeval      stamp;
    struct net_device   *dev;
    struct net_device   *real_dev;
```

The next three fields are unions that are expanded into specific head type fields. The purpose of these fields is to provide convenient access to all of the header fields in the packet. H points to the transport layer header in the packet contained by this buffer, and nh points to the network layer header in the packet contained by this buffer. Mac points to the link layer header.

```
    union {
```

These are the transport layer headers, but actually include any protocol that is encapsulated by IP. The first four headers are for TCP, UDP, ICMP, and IGMP, respectively.

```
        struct tcphdr   *th;
        struct udphdr   *uh;
        struct icmphdr  *icmph;
        struct igmphdr  *igmph;
```

Ipip is the IP tunneling header, and raw is a generic header pointer.

```
        struct iphdr    *ipiph;
        unsigned char   *raw;
    }  h;
    union {
```

These are the network layer headers. Iph is the IPv4 header, and ipv6h is the IPv6 header.

```
        sstruct iphdr    *iph;
        struct ipv6hdr  *ipv6h;
```

Arph is the ARP protocol header, and raw is the generic network layer header.

```
        struct arphdr     *arph;
        unsigned char   *raw;
    }  nh;
    union {

          struct ethhdr   *ethernet;
          unsigned char   *raw;
    }  mac;
```

The next field, dst, points to the destination address cache entry for this packet. It could also be the routing cache entry for the packet. We discuss the destination cache extensively elsewhere in the book.

```
    struct  dst_entry   *dst;
```

The next field, sec_path, is used with XFRM, the internal kernel implementation of IPSec.

```
    struct  sec_path    *sp;
```

The next field, cb, is called the control buffer. It is for private storage of protocol-specific items that must be copied across layers. In TCP/IP, it is primarily used by TCP for the TCP control buffer, which is discussed in Chapter 8. The contents of the cb field are copied to a new buffer when the sk_buff is cloned.0

```
    char                cb[48];
```

The next field, len, contains the overall length of the packet including the header and any data in attached buffers. Data_len is the length of the data part of the packet. Csum holds the IP header checksum.

```
    unsigned int        len,
                        data_len,
                        csum;
```

Local_df holds the "don't fragment" value. If set, this packet will not be fragmented when it is transmitted. Cloned indicates that this packet was cloned. Pkt_type is the class of the packet and is set to one of the values in Table 7.5. The field ip_summed indicates whether the output device can calculate IP checksums in hardware.

Table 7.5: Packet Types in Socket Buffer pkt_type Field

Packet Type	Value	Purpose
PACKET_HOST	0	Packet is directed to this machine.
PACKET_BROADCAST	1	Broadcast packet.
PACKET_MULTICAST	2	Multicast packet.
PACKET_OTHERHOST	3	Unicast packet for sending to a peer.
PACKET_OUTGOING	4	Outgoing packet.
PACKET_LOOPBACK	5	Any looped back broadcast or unicast packet.
PACKET_FASTROUTE	6	This packet is to be processed in the fast-route path.

```
    unsigned char       local_df,
                        cloned,
                        pkt_type,
                        ip_summed;
```

Priority is the packet queuing priority. Protocol is the value of the protocol field in the packet's MAC header. This field is used to dispatch a packet to the network layer protocol handler.

```
    __u32               priority;
    unsigned short      protocol,
                        security;
```

The next field points to the destructor function for this packet or NULL if there isn't any.

```
    void                (*destructor)(struct sk_buff *skb);
```

The following fields are used by netfilter if it is configured

```
#ifdef CONFIG_NETFILTER
        unsigned long       nfmark;
    __u32               nfcache;
    struct nf_ct_info   *nfct;
#ifdef CONFIG_NETFILTER_DEBUG
        unsigned int        nf_debug;
#endif
```

These fields are used by the bridge net filter.

```
#ifdef CONFIG_BRIDGE_NETFILTER
    struct nf_bridge_info *nf_bridge;
#endif
#endif /* CONFIG_NETFILTER */
#if defined(CONFIG_HIPPI)
    union {
        __u32           ifield;
    }  private;
#endif
```

This option field is the traffic control index.

```
#ifdef CONFIG_NET_SCHED
       __u32             tc_index;          /* traffic control index */
#endif
```

The following fields must be at the end of the sk_buff. This is necessary because of the way the function alloc_skb allocates a new socket buffer. Socket buffers may be followed by pages containing packet data, which are allocated directly from the kernel's paged memory. Truesize is the actual packet size. Users is the packet user count. This field must be implemented atomically.

```
    unsigned int        truesize;
    atomic_t            users;
```

Head points to the start of the attached data buffer. Data points to the beginning of useable data in the attached data buffer. Tail points to the end of the useable data in the attached buffer. End points to the absolute end of the data. It is also the start of the "shared" portion that is common to all clones of the buffer.

```
    unsigned char       *head,
                        *data,
                        *tail,
                        *end;
} ;
```

Table 7.5, shown earlier shows the packet classes in the pkt_type field. These classes are used by IP to determine how to internally route a packet. The values are set by decoding the packet's destination address and checking it in the packet's destination and routing caches.

7.7.2 Socket Buffer Explanation

On the transmit side of the TCP/IP stack, socket buffers are generally allocated in the socket layer before data is ready to transmit. However, socket buffers may be allocated anywhere in the protocol stack. Buffers are allocated whenever any internal protocol in the stack needs to send a packet. This can happen within TCP, ARP, IGMP, or elsewhere. If the packet survives error checking and other processing, the socket buffer is not de-allocated until the driver's hard_start_xmit network returns, which indicates that the packet was actually physically transmitted. For more details on how a packet is transmitted through a network interface driver, refer to Chapter 4, "Linux Networking Interfaces and Device Drivers." On the receive side, socket buffers are allocated by the network interface driver's interrupt service routine as a new packet is received from the physical interface. If the packet is not de-allocated as a result of error checking and processing, it remains until the application code receives the data, and then the socket buffer is de-allocated. If the packet is dropped for any reason, the socket buffer is de-allocated.

The first three fields in the socket buffer are for list maintenance, including the two pointers next and prev. These two pointers are identical to the fields in the sk_buff_head so pointers can be casted, making it easier to process lists and queues of sk_buffs. The third field, list, points back to the socket buffer head. Lists of socket buffers can exist for many reasons and may also be overloaded with similar types such as TIME-WAIT buckets. This is why the socket buffer fields involved with list maintenance are generic and placed in the beginning of the structure. Linux allocates socket buffers from a slab cache called the skbuff_head_cache. The slab allocator is described earlier in this chapter. When buffers are de-allocated, they are returned to the slab cache. The next field, sk, is relevant for buffers that originate from a transport layer or the socket layer. It points to the socket structure (struct sock) of an open socket if there is one associated with the buffer. The sk field can be de-referenced anywhere in the protocol stack to access the file descriptor, IO- related parameters, and status associated with the open socket. If the socket buffer contains a received packet, the next field, stamp, is set by the device driver to the time when the packet was received at the interface. The field dev is also used for received packets. It points to the receiving driver's network device structure. The h, nh, and mac fields are pointers to the transport, network, and link layer header fields of a TCP/IP packet. The dst field points to the destination cache entry. As discussed elsewhere in this book, this cache entry can be either the destination cache or neighbor cache. These caches may include information about the hardware address of the next system to receive this packet, routing information, or internal destination information. The neighbor cache and destination caches are described in more detail in Chapter 6, "The Linux TCP/IP Stack." The cb buffer is a private area available to socket buffer users to use as they wish. However, it is generally used by the TCP protocol to contain the TCP control buffer. It is important to note that this buffer is explicitly copied to the new sk_buff when a socket buffer is cloned. The len field contains the length of the packet. The value of len includes both the packet header and the data.

The data_len field contains the length of the packet data only without the header and is used by the IP fragmentation and de-fragmentation facility. When a packet has a list of fragments, the frag list is constructed from a chain of socket buffers consisting of a head skb followed by a list of sk_buffs, each of which points to a single fragment. In this case, the data_len field has the sum of the length of all the fragments without the IP headers. See Chapter 9, "The Network Layer, IP," for detailed information about IP fragmentation. The users field is incremented whenever a new reference is generated to the skb. Sharing a socket buffer increments users, but cloning a socket buffer does not. The cloned field is set whenever a buffer is cloned.

The final socket buffer fields—head, tail, end, and data—require a little more explanation. The head pointer points to the start of the packet, and tail points to the end of the packet. The end field points to the absolute end of the user data portion of the packet. The application program, device driver, or even the internal protocol stack layers will rarely change these fields directly. Generally, they are accessed through the socket buffer utility functions listed in Section 7.6. As discussed earlier, one of the requirements for any networking buffer system is that headers can be pre-pended or removed without copying the packet contents. A TCP/IP packet grows in length from where it was formed at the socket layer and traverses down the stack to the network interface drivers. The growth occurs as each layer encapsulates the data it receives from the layer above with its own header. Usually, the growth is at the beginning of the packet as headers are put in front of the data received from layers above. It is important to reduce or eliminate the physical copying packets when doing this header manipulation and far better to do the encapsulation by moving pointers. When a socket buffer is allocated at the socket layer, sufficient space is reserved for the total maximize size or MTU of the interface likely to transmit the packet. This is called *reserving headroom*. Then, as the socket buffer travels down the stack, packet headers can be placed in front without copying the data using a set of utility functions that use the head, tail, end, and data fields. These fields are provided to enable the API functions to keep track of positions in the buffer where data of interest starts and ends, and these functions use these pointers.

7.8 Socket Buffers, Fragmentation and Segmentation

The shared info structure, skb_shared_info, is used to support IP fragmentation and TCP segmentation. A discussion of the socket buffers is not complete without discussing this structure. The shared info structure, also known as skb_shinfo, is defined in the file *include/linux/skbuff.h*.

```
struct skb_shared_info {
```

This field contains the reference count for this skb. It is incremented each time the buffer is cloned.

```
    atomic_t            dataref;
```

Nr_frags is the number of fragments in this packet. This field is used by TCP segmentation.

```
    unsigned int        nr_frags;
```

The next two fields are used for devices that have the capability of doing TCP segment processing in hardware. This is the network device feature, NET_F_TSO.

```
    unsigned short      tso_size;
    unsigned short      tso_segs;
```

This field points to the list of fragments for this packet if it is fragmented.

```
    struct sk_buff      *frag_list;
```

This is the array of page table entries. Each entry is actually a TCP segment.

```
    skb_frag_t          frags[MAX_SKB_FRAGS];
} ;
```

This structure is placed at the end of the attached data buffer and pointed to by the end field in the socket buffer structure, which points to the end of the data portion of the packet. However, end is also used to find the beginning of skb_shinfo in the attached data buffer because it immediately follows the data portion of the regular packet. Skb_shared_info has several purposes, including IP fragmentation, TCP segmentation, and keeping track of cloned socket buffers. When used for IP fragmentation, skb_shared_info points to a list of sk_buffs containing IP fragments. When used for TCP segmentation, this structure contains an array of attached pages containing the segment data. The handling of TCP segments is more efficient than IP fragments. IP fragmentation is not quite as common as it was in earlier days of the Internet. Fragmentation is used when a network segment has a smaller MTU than the packet size. IP fragmentation is necessary if the MTU of the outgoing device is smaller than the packet size. See Chapter 9 for more details about IP fragmentation. TCP segmentation, however, is far more common because it is the underlying mechanism for the transport of streaming data that occurs in most network traffic

Skb_shinfo can also be used to hold TCP segments. When used this way, skb_shinfo contains an array of pointers to memory mapped pages containing TCP segments. TCP provides a streaming service that makes the data look like an uninterrupted sequence of bytes even though the data must be split up to fit into IP packets. See Chapter 8, "Sending the Data from the Socket through UDP and TCP," for more information about TCP segmentation. When a socket buffer is cloned, skb_shared_info is copied to the new buffer.

The first field in the shared info structure, dataref, indicates that a socket buffer is cloned if the value is nonzero because it is incremented each time a socket buffer is cloned. (The cloned field in the socket buffer is also set to one when a socket buffer is cloned.) The next field in the shared info structure, frag_list, is used by the IP fragment reassembly facility. This is how each fragment on the list can share the same IP header. The IP headers for each fragment are almost identical. They differ only in the fragment ID field, the fragment offset, and the checksum. When the input processing in the IP protocol discovers that an incoming skb is actually an IP fragment, it places the packet on a special list containing the fragments. IP moves this list (without copying the actual packet data) into a single datagram consisting of a head socket buffer followed by a list of socket buffers, each of which points to a single fragment. The frag_list field in the shared info area points to the list of socket buffers containing the fragments. Although each IP fragment occupies a separate socket buffer, the skb_shinfo structure itself is copied to each socket buffer when it is created. See Figure 7.5 for an illustration of a socket buffer that points to an array of IP fragments.

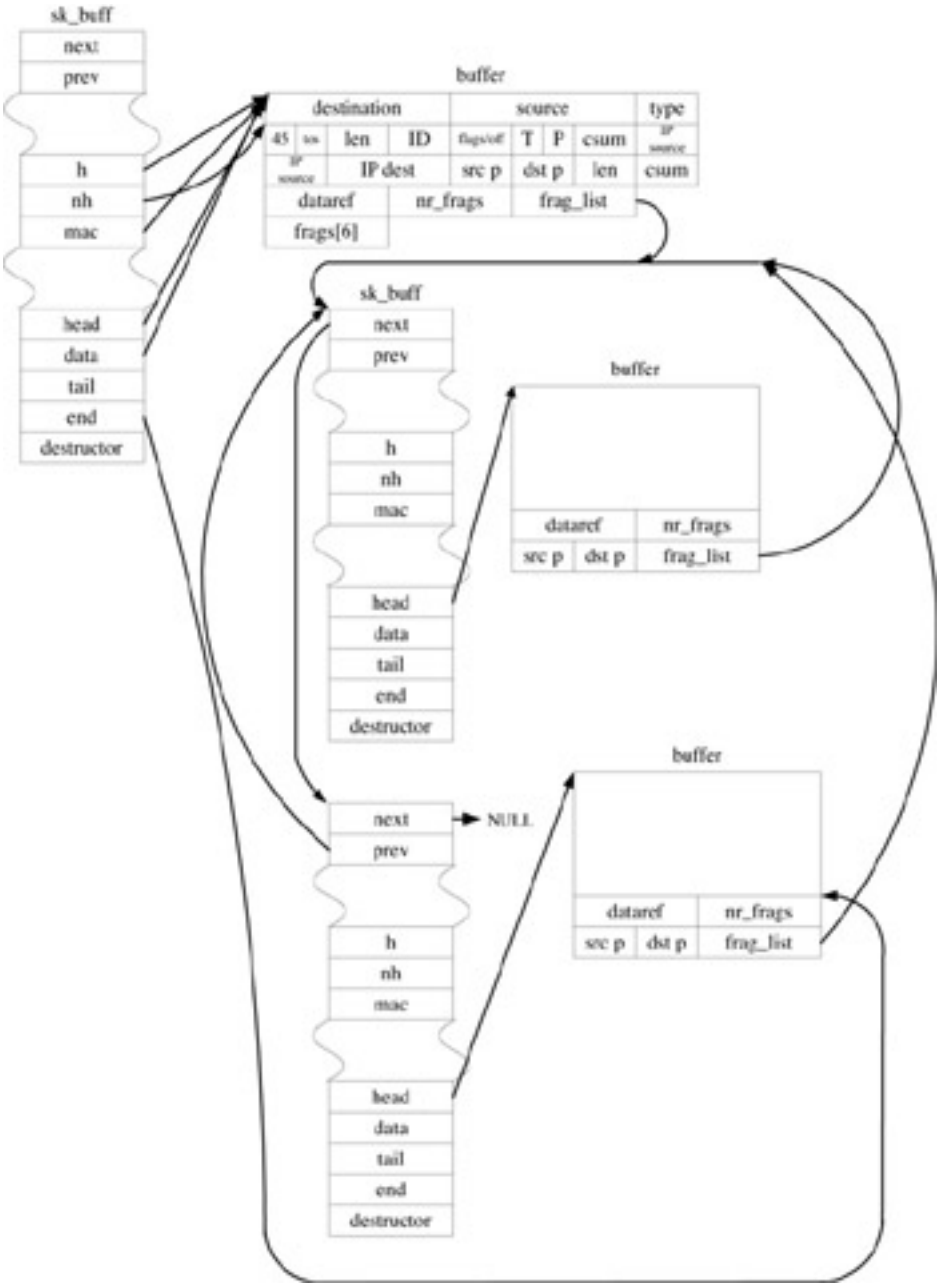

Figure 7.5: Sk_buff with fragments.

The second field in the shared info structure, nr_frag, is not used for IP fragmentation; instead, it is for TCP segmentation. A socket buffer containing segments is indicated when this field contains a nonzero value. The value of nr_frag corresponds to the number of segment pages attached to the socket buffer, and the shared_info structure contains pointers to the segment pages in the field frags. The array of frags is placed in memory immediately after the nr_frag field. It can contain as many as six pages in the array. The actual number of locations in the array will depend on the hardware architecture and the configured page size, PAGE_SIZE. Each of the elements in the frags array points to a memory-mapped page in the Linux virtual page table array. Refer to Section 7.4 for more information about Linux slab allocation and virtual page tables. When a socket buffer created by TCP contains a chain of segments, each sequential segment's data is in a separate memory mapped page pointed to by a location in the frags array. This is

considerably more efficient than maintaining a redundant sk_buff structure for each TCP segment. Once TCP is in the ESTABLISHED state, the packet headers for subsequent segments are nearly identical so the packet header can be shared among each of the segments. Processing time is saved during processing by not requiring Linux to copy a complete IP header for each segment.

Each location in the frags array consists of the skb_frag_t structure defined in file *linux/include/linux/skbuff.h*. The size of the frags array is calculated to hold a total of 64 Kbytes of data.

```
#define MAX_SKB_FRAGS (65536/PAGE_SIZE + 2)
typedef struct skb_frag_struct skb_frag_t;
struct skb_frag_struct {
```

Page is a pointer to a page table entry. The next field, offset, is the offset from the start of the page to where the data begins. Size is the length of data in page.

```
    struct page   *page;
    __u16         page_offset;
    __u16         size;
} ;
```

Figure 7.6 is an illustration of a socket buffer containing an array of TCP segments.

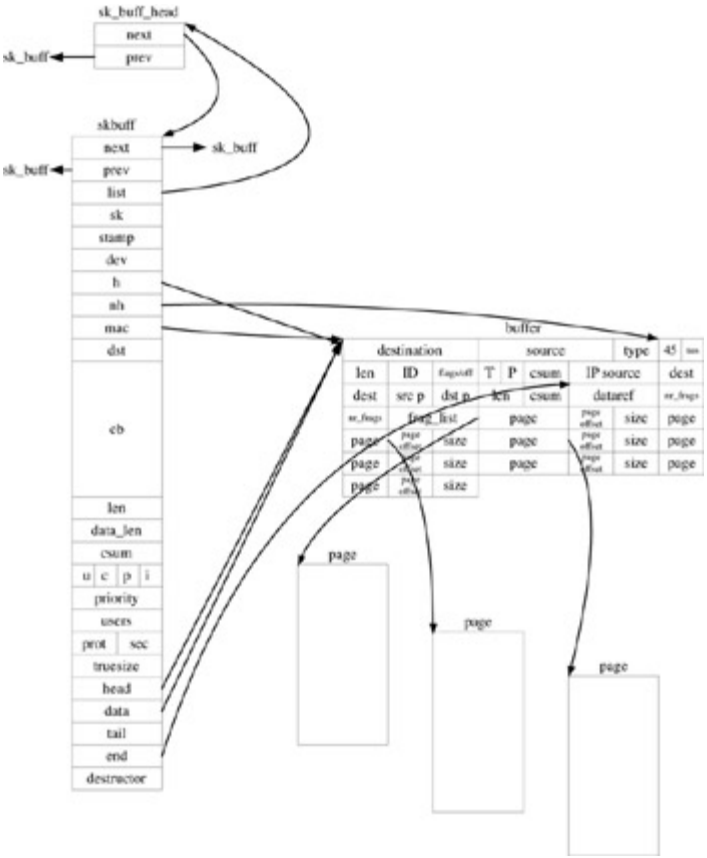

Figure 7.6: Sk_buff with segments.

7.9 Socket Buffer Allocation and Lists

As discussed in Section 7.4, the Linux slab memory allocation system is suitable for networking protocol stacks because it is efficient and is not prone to fragmentation. The socket buffer header cache also provides a performance enhancement because the socket buffers are not returned to the general kernel memory when they are freed, but instead are placed on a separate header cache. There are multiple header caches, one for each CPU in multiple processor implementations. The buffers are fetched or returned to the header cache for the current CPU. When a sk_buff is created, the allocation function, alloc_skb, allocates socket buffers from the slab cache, skbuff_head_cache. As we saw earlier, if the slab cache does not have any socket buffers, it is grown, by adding pages from the kernel's general memory allocation slab. When a caller frees a socket buffer, the socket buffer is returned to the header cache pool. The skbuff_head_cache consists of an array of socket buffer header structures indexed by CPU number. The fields of the socket buffer header structure, next and prev, correspond to the first two fields of the sk_buf.

7.9.1 Socket Buffer Cloning

When a sk_buff is cloned, the skb_shinfo is replicated. Cloning allows TCP fragment arrays to share the same socket buffer structure and packet header. Cloned buffers are not the same as shared buffers. Although confusing, it is important to differentiate the two. A shared socket buffer is held by more than one user. Each time the buffer is referenced by another user, the users field is incremented, and each time it is de-referenced, the field is decremented. A cloned skb is something entirely different. This is confusing because the term *shared* is often used in the context of buffer cloning. A cloned sk_buff points to the same data as the sk_buff from which it was cloned. It has most of the same fields set as the original buffer. In addition, it has a copy of the cb field from the original sk_buff. Most important, it has the same attached data buffer, usually consisting of an IP header and the skb_shinfo, which immediately follows the data. The original and the clone both contain a list of page table pointers, each of which points to a page with a TCP segment. Figure 7.6 illustrates a cloned buffer with an array of TCP segments.

7.9.2 Socket Buffer Queues

As we saw earlier, socket buffers contain the next and previous pointers as the first fields. This is done to facilitate the maintenance of queues. The ability to manipulate queues of network buffers is of fundamental requirement of any buffer system used in networking. For this reason, Linux provides a group of interface functions for en-queuing, de-queuing, and maintaining lists of socket buffers. For an example of the use of queues of sk_buffs, we can look at Chapter 6, which contains more detailed information on the queuing layer.

7.10 Socket Buffer Utility Functions

Linux provides many utility functions to allocate, de-allocate, and manipulate the socket buffers, sk_buffs. Some of the socket buffer functions are not interrupt safe unless used properly, and generally, there are both safe and unsafe versions of these functions. The unsafe functions include the parameter gfp_mask, where GFP stands for "get free pages." Gfp_mask must be set

to GFP_ATOMIC if the functions are called from an interrupt service routine. However, interrupt safe versions of most of the functions are declared in *linux/include/linux/sk_buf.h*. If possible, it is preferable to call the safe versions because the unsafe versions are used only where the caller has already masked interrupts. Most of the functions listed later in this section are the safe versions. The first set of functions controls socket buffer allocation and de- allocation. The next group of functions is for copying and cloning socket buffers. Another group of functions is for the manipulation of the data pointers in the socket buffer to make room for headers in the front of a packet or extend the end of a packet. The final group of socket buffer calls includes functions for manipulating lists and queues of socket buffers. This section explains each of the utility functions in detail.

7.10.1 Functions for Socket Buffer Allocation and De-Allocation

The first function, alloc_skb, allocates a new socket buffer.

```
extern struct sk_buff *alloc_skb(unsigned int size, int priority);
```

This function is unsafe. It must be called with gfp_mask value set to GFP_ATOMIC if it is called from an interrupt service routine. It is called with gfp_mask set to GFP_KERNEL when called from elsewhere in the Linux kernel. The first thing alloc_skb does after checking gfp_mask is try to get the skb from the socket buffer header list by calling skb_head_from_pool. If there are no more sk_buffs available, it allocates a new one by calling kmem_cache_alloc to get the buffer from the slab cache. Every skb must point to at least one data buffer, so a new data buffer is always allocated from the kernel's general slab cache by calling kmalloc. The data buffer must contain at least enough space for the packet header and the skb_shinfo structure, so kmalloc requests an amount of bytes equal to or greater than the value of the size parameter, plus the length of the skb_shinfo structure. Alloc_skb aligns size according to the machine's architecture, so if the caller requests an odd size, it will be rounded up.

Next, a few fields in the socket buffer structure are set. Since the socket buffer is allocated from the slab cache, most fields have been pre-initialized. The truesize field is set to the length of the attached buffer, plus the length of the socket buffer structure itself. The head, data, and tail pointers are all set to point to the beginning of the actual data buffer, which was allocated with kmalloc. End is set to point to the end of the user data part of the buffer, and as discussed earlier, this is where the skb_shinfo structure lives. The len, cloned, and data_len fields are all set to zero. These fields are adjusted later to reflect changes in the size of the packet pointed to by this sk_buff. The users field is set to one to indicate that this is the first reference to the sk_buff. Finally, a few fields are initialized in the skb_shinfo structure. Because there is neither a fragment list yet or an array of frag pages defined for this packet, dataref is set to one, and frag_list is set to NULL.

Dev_alloc_skb also allocates a new socket buffer, sk_buff.

```
static inline struct sk_buff *dev_alloc_skb(unsigned int length);
```

Dev_alloc_skb is provided for convenience. It is used by Ethernet network interface drivers and other network interface drivers. This function is safe, and internally it calls alloc_skb with the gfp_mask set to GFP_ATOMIC. In addition, it adds 16 bytes to the length field to reserve room

for the Ethernet MAC header before calling alloc_skb to actually allocate the sk_buff and the attached data buffer.

Kfree_skb frees a socket buffer.

```
static inline void kfree_skb(struct sk_buff *skb);
```

Before freeing the buffer, it checks the use count in the users field, and if the use count is one, it goes ahead and frees the buffer. Before freeing, it checks to make sure that the skb was removed from all lists by checking the list field, and if list is NULL, it means that the sk_buff is not a member of anybody's list. Next, it releases the destination cache entry. This will return the destination cache entry back to the destination cache slab. For details about the destination cache and the neighbor cache system, see Chapter 6. It calls the destructor function if the destructor field is not equal to NULL. Next, it cleans up the socket buffer state by setting most of the fields to zero. This prepares the buffer for returning to the slab cache so it can be used later without much initialization. The attached buffer must be freed as well, and after freeing the attached buffer, it calls skb_head_to_pool to return the cleaned-up sk_buff to the header cache.

7.10.2 Functions to Copy and Clone Socket Buffers

All of the functions in this group copy or clone socket buffers in various ways. Skb_cow copies the socket buffer and expands its headroom.

```
int skb_cow(struct sk_buff *skb, unsigned int headroom);
```

Skb_cow copies the header of the skb if required. It expands the headroom at the start of the buffer, which is the space between the start of the buffer and the point in the buffer referenced to by the data field in skb. If the socket buffer does not have enough headroom or its data part is shared, the data is re-allocated and both the skb header data and shared info are copied to the new buffer. The frags and frag_list are copied to the skb_shinfo structure in the new skb. In the new socket buffer, pointers are adjusted to point to the correct corresponding places in the attached data buffer. Skb_cow does not expand the headroom by less than 16 bytes, so headroom is rounded up to the next largest multiple of 16.

Skb_clone duplicates a socket buffer.

```
struct sk_buff *skb_clone(struct sk_buff *skb, int priority);
```

It allocates a new socket buffer structure and returns a pointer to the new sk_buff. This call is not safe unless it is called with the gfp_mask set to GFP_ATOMIC or GFP_KERNEL. The prev, next, and list fields in the new sk_buff are set to NULL because the new socket buffer is not a member of any list. The destructor and sk fields are set to NULL because the new buffer is not associated with a socket. The users field in the new buffer is set to one. It copies the other fields from the old socket buffer to the new one. After cloning, the new sk_buff points to the same attached data buffer as the old one. The dataref field in the skb_shinfo area of the attached buffer is incremented to indicate that there is an additional reference to the attached data buffer.

Skb_cloned checks to see if a socket buffer is a clone.

```
static inline int skb_cloned(const struct sk_buff *skb);
```

It returns TRUE if the dataref field in the skb_shinfo structure in the attached buffer is greater than one and if the cloned field is set in skb.

Skb_copy copies a socket buffer.

```
struct sk_buff *skb_copy(const struct sk_buff *skb, int priority);
```

The function copies the socket buffer structure, skb, along with the attached data buffer. It returns a pointer to the new socket buffer structure. The new sk_buff will be given an attached buffer with enough space to hold all the data in skb. A new shared info structure is initialized in the buffer attached to the new sk_buff. In addition, the data from any attached fragment list or fragment pages is copied directly into the new buffer. The new buffer is an entirely new copy, so the caller may modify any data pointed to by the new buffer. Header space is reserved in the new buffer by adjusting the data pointer.

Pskb_copy copies a socket buffer with a private header portion.

```
struct sk_buff *pskb_copy(struct sk_buff *skb, int gfp_mask);
```

A socket buffer and the header part of the attached data buffer are copied into a new sk_buff. A pointer to the new sk_buff is returned. This function does not copy the fragmented portion of the original socket buffer. It copies skb_shinfo only so the fragment list and segment array are shared with the old socket buffer. Header space is reserved in the new buffer by adjusting the data pointer.

The function skb_shared checks to see if a socket buffer is shared.

```
int skb_shared(const struct sk_buff *skb);
```

It checks the reference count of the socket buffer by looking at the users field in skb. It returns TRUE if users is not equal to one.

Skb_share_check checks if a socket buffer is shared and clones the buffer if it is.

```
struct sk_buff *skb_share_check(struct sk_buff *skb, int pri);
```

First, this function checks to see if a socket buffer is shared. If it is shared, skb_share_check clones the socket buffer by calling skb_clone. It drops the reference count to the old socket buffer by one by calling kfree_skb.

Skb_get gets a new reference to a socket buffer.

```
struct sk_buff *skb_get(struct sk_buff *skb);
```

Skb_get increments the reference count by adding one to the users field of skb. This indicates that the socket buffer is shared. It returns a pointer to skb.

7.10.3 Functions to Manipulate Socket Buffer Pointer Fields

In this section, we discuss functions provided to pre-pend headers in the front of packets or remove headers from a packet in addition to functions for manipulating the amount of space at the end of the packet. All the functions in this group operate without copying the data.

The first of these functions, skb_headroom reserves space at the front of the attached buffer.

```
int skb_headroom(const struct sk_buff *skb);
```

It returns the amount of available space between the start of the attached buffer and the start of the data.

The function skb_put extends the user data area of the socket buffer by len.

```
unsigned char *skb_put(struct sk_buff *skb, unsigned int len);
```

It extends the user data area of the attached buffer by adjusting the tail and len fields in skb. The entire new length must not exceed the total buffer size.

Skb_tailroom gets the amount of space at the end of the buffer.

```
int skb_tailroom(const struct sk_buff *skb);
```

This function gets the amount of space at the end of the attached buffer pointed to by skb. It subtracts the value of tail from the value of end in the socket buffer, skb. If skb is not linear, skb_tailroom returns zero. A nonlinear sk_buff is one with attached fragments. A sk_buff with attached fragments has no user data in the attached buffer.

Skb_reserve adjusts the headroom of a socket buffer, skb, by length len.

```
void skb_reserve(struct sk_buff *skb, unsigned int len);
```

It adjusts the headroom of the data buffer attached to skb. It increases the data and tail fields in skb by the desired length.

Skb_push adds data space to the front of socket buffer, skb.

```
unsigned char *skb_push(struct sk_buff *skb, unsigned int len);
```

Skb_push adds data space to the front of a socket buffer by changing the data field in skb to point to the new start of the useable data. It increases len in skb by the new amount of space. It returns a pointer to the new start of the useable data.

Skb_pull removes the amount of data space specified by len from the front of a socket buffer.

```
unsigned char *skb_pull(struct sk_buff *skb, unsigned int len);
```

It removes the data space from the front of the data buffer and advances the data field in skb to point to the new start of the useable data. It subtracts the number of bytes of removed space from the len field. It returns a pointer to the new start of the useable data.

Skb_trim removes space from the end of socket buffer, skb.

```
void skb_trim(struct sk_buff *skb, unsigned int len);
```

Skb_trim reduces the length of the buffer attached to the socket buffer by setting the len field in skb to the length argument and adjusting the tail field to point to the beginning of the data, plus length.

7.10.4 Functions to Manage Lists of Socket Buffers

The functions in this group are provided by Linux for the use of writers of protocol and network interface drivers to manage socket buffer lists. Queues of socket buffers are a key part of all the layers and member protocols in Linux TCP/IP. In each of the following functions, the structure sk_buff_list is used to hold the head of a list of sk_buffs. Every function in this group is safe to call from anywhere in the code because all the functions acquire irq_spinlock before manipulating the list of socket buffers.

The first function, skb_queue_head, places the socket buffer, newsk, at the beginning of a list pointed to by list.

```
void skb_queue_head(struct sk_buff_head *list, struct sk_buff *newsk);
```

The next function, skb_queue_tail, puts the socket buffer, newsk, at the end of a list.

```
void skb_queue_tail(struct sk_buff_head *list, struct sk_buff *newsk);
```

Skb_dequeue removes the sk_buff from the beginning of the list, pointed to by list. It returns a pointer to the removed socket buffer.

```
struct sk_buff *skb_dequeue(struct sk_buff_head *list);
```

The function skb_dequeue_tail removes a socket buffer from the end of a list and returns a pointer to the removed item.

```
struct sk_buff *skb_dequeue_tail(struct sk_buff_head *list);
```

Skb_append inserts the socket buffer, newsk, anywhere on a list of socket buffers. It puts the new socket buffer after the one pointed to by old.

```
void skb_append(struct sk_buff *old, struct sk_buff *newsk);
```

Skb_insert also inserts a socket buffer on a list of sk_buffs. It puts the new sk_buff on the list ahead of the socket buffer pointed to by old.

```
void skb_insert(struct sk_buff *old, struct sk_buff *newsk);
```

This function, skb_queue_empty, checks to see if a list of socket buffers is empty. It does this by checking if the next field in skb points back to itself. If the list is empty, it returns TRUE; otherwise, it returns FALSE.

```
int skb_queue_empty(const struct sk_buff_head *list);
```

Skb_queue_len returns the length of a list of socket buffers by checking the qlen field in the sk_buff head structure.

```
__u32 skb_queue_len(const struct sk_buff_head *list_);
```

Skb_queue_purge removes all socket buffers on a list. Each socket buffer on the list is dropped by calling kfree_skb.

```
void skb_queue_purge(struct sk_buff_head *list);
```

Skb_peek_tail returns a pointer to the last skb on the list.

```
struct sk_buff *skb_peek_tail(struct sk_buff_head *list_);
```

Finally, skb_queue_head_init initializes a queue. It sets up the queue header as well as the next and prev pointers in the sk_buff_head structure pointed to by list.

```
static inline void skb_queue_head_init(struct sk_buff_head *list);
```

7.11 Some Practical Considerations

Slab cache allocation statistics are kept in the /proc file system. They can be examined by executing the command cat/proc/slabinfo. Refer to the man page, slabinfo(5) for more information. Table 7.6 shows the statistics for slab caches used by the TCP/IP stack that were active at one point in the author's workstation.

Table 7.6: Active Slab Caches

Cache_name	**# of Active**	**# Available**	**Size**	**# Pages**	**Total # Pages**	**# Pages per Slab**
ip_fib_hash	14	113	32	1	1	1
clip_arp_cache	0	0	128	0	0	1
ip_mrt_cache	0	0	96	0	0	1
tcp_tw_bucket	0	30	128	0	1	1
tcp_bind_bucket	11	113	32	1	1	1
tcp_open_request	0	0	96	0	0	1

Table 7.6: Active Slab Caches

Cache_name	# of Active	# Available	Size	# Pages	Total # Pages	# Pages per Slab
inet_peer_cache	1	59	64	1	1	1
ip_dst_cache	100	180	192	8	9	1
arp_cache	4	30	128	1	1	1
kmem_cache						

7.12 Summary

In this chapter, we covered Linux network memory allocation and the Linux socket buffer structure. Socket buffers are the most fundamental part of Linux network memory allocation and they should be examined for a complete understanding of the Linux TCP/IP protocol suite. We began by discussing various memory allocation schemes used for allocating packet buffers in network protocol stacks. We contrasted the Linux slab allocation method with other allocation schemes and compared the advantage with slab allocation for efficiency and avoiding memory fragmentation. In addition, we discussed Linux socket buffers, sk_buffs, the primary structure used in Linux to hold network packets. We discussed socket buffer cloning and copying and listed functions for manipulating socket buffer headers and tail space without copying the data. In addition, we covered types of socket buffers and how they can be formed into lists and queues.

Chapter 8: Sending the Data from the Socket through UDP and TCP

We began this book by looking at the fundamentals of networking and the basics of the TCP/IP protocol suite. Most of the book focused on the internal structure of Linux. The middle chapters covered network interface drivers, Linux sockets, Linux infrastructure, and Linux socket buffers. This chapter continues the discussion of the internals of Linux TCP/IP. It covers the implementation of the transport layer protocols, UDP and TCP. Often, the best way to understand a complex system of protocols is to follow the data as it flows through the system. Remember that TCP/IP is a layered protocol implemented as a stack. For transmission, data is prepared when the application writes it into the socket at the top of the stack. It flows down through the transport protocol on to the network layer protocol and out the physical interface. In this chapter, we explain what happens inside the transport protocols by watching the data as it flows from the socket to the IP layer. We follow a packet as it is sent from a socket through either UDP or TCP and arrives at the IP protocol.

8.1 Introduction

In this chapter, we look at what happens in the transport layer as data is transmitted. When a user writes data into an open socket, socket buffers are allocated by the transport layer and travel through the transport layer to IP where they are routed and passed to the device drivers for sending. Specifying SOCK_DGRAM in the socket call invokes the UDP protocol, and specifying SOCK_STREAM invokes the TCP protocol. For SOCK_DGRAM type sockets, the process is relatively simple, but it is far more complicated for SOCK_STREAM type sockets. We will examine both UDP and TCP and look at the functions that interface the protocol to the socket layer. Next, we will focus on the sendmsg function for each of the protocols. We will follow the data as it flows through the transport layer.

8.2 Socket Layer Glue

Before we start looking at the internals of each of the transport layer protocols, we should look at how service functions in the transport layer protocols are associated with the socket layer functions. Through this mechanism, the application program is able to direct the actions of the transport layer for each of the socket types, SOCK_STREAM and SOCK_DGRAM. As explained in Chapter 5, "Linux Sockets," each of the two transport protocols is registered with the socket layer.

8.2.1 The Proto Structure

The key to this registration process is the data structure, proto, which is defined in *linux/include/linux/sock.h*. Most of the fields in the proto structure are function pointers. They each correspond to specific functions for each transport protocol. A transport protocol does not have to implement every function; for example, UDP does not have a shutdown function. UDP and TCP do implement most of the functions in the proto structure. The first seven functions, close through shutdown, are described in Sections 8.2.3 for UDP and 8.2.4 for TCP.

```
struct proto {
    void          (*close)(struct sock *sk,
                           long timeout);
    int           (*connect)(struct sock *sk,
struct sockaddr *uaddr, int addr_len);
    int           (*disconnect)(struct sock *sk, int flags);
struct sock *     (*accept) (struct sock *sk, int flags, int *err);
    int           (*ioctl)(struct sock *sk, int cmd,
                           unsigned long arg);
    int           (*init)(struct sock *sk);
    int           (*destroy)(struct sock *sk);
    void          (*shutdown)(struct sock *sk, int how);
```

The getsockopt and setsockopt functions and options for both UDP and TCP are discussed in detail later in this section.

```
    int           (*setsockopt)(struct sock *sk, int level,
                                int optname, char *optval,
                                int optlen);
    int           (*getsockopt)(struct sock *sk, int level,
                                int optname, char *optval,
                                int *option);
```

The sendmsg function is discussed in Section 8.5 for UDP and Section 8.6 for TCP.

```
    int           (*sendmsg)(struct sock *sk,
                             struct msghdr *msg, int len);
```

Recvmsg is covered in Chapter 10, "Receiving the Data in the Transport Layer, UDP and TCP."

```
    int            (*recvmsg)(struct sock *sk,
                              struct msghdr *msg, int len,
                              int noblock, int flags,
                              int *addr_len);
```

The bind function is not implemented by either TCP or UDP within the transport protocols themselves. Instead, it is implemented at the socket layer, covered in Chapter 4, "Linux Sockets and Socket Layer Programming."

```
    int           (*bind)(struct sock *sk,
struct sockaddr *uaddr,int addr_len);
```

Backlog_rcv is implemented by TCP. Refer to Chapter 10, "Receiving Data in the Transport Layer, UDP and TCP," to see what happens when backlog_rcv is executed.

```
    int           (*backlog_rcv) (struct sock *sk,
                                  struct sk_buff *skb);
```

The hash and unhash functions are for manipulating hash tables. These tables are for associating the endpoint addresses (port numbers) with open sockets. The tables map transport protocol port numbers to instances of struct sock. Hash places a reference to the sock structure, sk, in the hash table.

```
    void          (*hash)(struct sock *sk);
```

Unhash removes the reference to sk from the hash table.

```
    void          (*unhash)(struct sock *sk);
```

Get_port returns the port associated with the sock structure, sk. Generally, the port is obtained from one of the protocol's port hash tables.

```
    int            (*get_port)(struct sock *sk,
                               unsigned short snum);
```

This field contains the name of the protocol, either "UDP" or "TCP".

```
    char            name[32];
    struct {
        int inuse;
        u8  __pad[SMP_CACHE_BYTES - sizeof(int)];
    }  stats[NR_CPUS];
} ;
```

Neither UDP nor TCP implement all of the functions in the proto structure. As we saw in Chapter 5, the AF_INET family provides pointers to default functions that get called from the socket layer in the case where the specific transport protocol doesn't implement a particular function. Each of the transport protocols registers a set of functions by initializing a data structure of type struct proto, defined in the file *sock.h*.

8.2.2 The Msghdr Structure

All the socket layer read and write functions are translated into calls to either rcvmsg or sendmsg, a BSD type message communication method. Internally in the socket layer, the internal functions use the msghdr structure, defined in file *linux/include/linux/socket.h*, to pass data to and from the underlying protocols.

```
struct msghdr {
```

Msg_name field is also known as the socket "name" or the destination address for this message. Generally, this field is cast into a pointer to a sockaddr_in. The msg_namelen field is the address length of the msg_name.

```
    void *        msg_name;
    int           msg_namelen;
```

Msg_iovec points to an array of data blocks passed either to the kernel from the application or from the kernel to the application. Msg_iovlen holds the number of data blocks pointed to by msg_iov. The msg_control field is for the BSD style file descriptor passing. Msg_controllen is the number of messages in the control message structure.

```
    struct iovec *      msg_iov;
    __kernel_size_t     msg_iovlen;
```

```
    void  *             msg_control;
    __kernel_size_t     msg_controllen;
    unsigned            msg_flags;
} ;
```

8.2.3 UDP Socket Glue

As we saw in Chapter 5, the transport protocols register with the socket layer by adding a pointer to a proto structure. UDP creates an instance of struct proto at compile time in the file *linux/net/ipv4/udp.c* and initializes it with values from Table 8.1.

Table 8.1: Protocol Block Functions for UDP, Struct proto

Protocol Block Structure Field Name	**UDP Function**
close	udp_close
connect	udp_connect
disconnect	udp_disconnect
ioctl	udp_ioctl
setsockopt	udp_setsockopt
getsockopt	udp_getsockopt
sendmsg	udp_sendmsg
recvmsg	udp_recvmsg
backlog_rcv	udp_queue_rcv_skb
hash	udp_v4_hash
unhash	udp_v4_unhash
get_port	udp_v4_get_port

The UDP protocol is invoked when the application layer specifies SOCK_DGRAM in the type field of the socket call. SOCK_DGRAM type sockets are fairly simple. There is no connection management or buffering. A call to one of the send functions in the application layer causes the data to be sent out immediately as a single datagram. Table 8.1 shows the UDP protocol functions mapped to each of the fields in the proto structure described earlier.

8.2.4 TCP Socket Glue

Like UDP, TCP registers a set of functions with the socket layer. As in UDP, this is done at compile time by initializing tcp_prot with the functions shown in Table 8.2 in the file *linux/net/ipv4/tcp_ipv4.c*. Tcp_prot is an instance of the proto structure and is initialized with the function pointers shown in Table 8.2.

Table 8.2: Protocol Block Functions for TCP, Struct proto

Protocol Block Structure Field Name	**TCP Function**
close	tcp_close

Table 8.2: Protocol Block Functions for TCP, Struct proto

Protocol Block Structure Field Name	TCP Function
connect	tcp_v4_connect
disconnect	tcp_disconnect
accept	tcp_accept
ioctl	tcp_ioctl
init	tcp_v4_init_sock
destroy	tcp_v4_destroy_sock
shutdown	tcp_shutdown
setsockopt	tcp_setsockop
getsockopt	tcp_getsockop
sendmsg	tcp_sendmsg
recvmsg	tcp_recvmsg
backlog_rcv	tcp_v4_do_rcv
hash	tcp_v4_hash
unhash	tcp_unhash
get_port	tcp_v4_get_port

8.2.5 Socket Options for TCP

In general, TCP is very configurable. The discussion of the internals of the TCP protocol later in this chapter and in Chapter 10 refer to various options and how they affect the performance or operation of the protocol. Section 8.5.2 shows the TCP options structure that holds the values of many of the socket options. However, in this section, the TCP socket options and ioctl configuration options are gathered together in one place. Although, most of these are covered in some fashion in the tcp(7) man page, this section lists applicable internal constants and internal variables as well as any references to other sections in the text.

The following options are set via the setsockopt system call or read back with the getsockopt system call.

TCP_CORK: If this option is set, TCP doesn't send out frames until there is enough data to fill the maximum segment size. It allows the application to stop transmission if the route MTU is less than the Minimum Segment Size (MSS). This option is unique to Linux, and application code using it will not be portable to other operating systems (OSs). This option is held in the nonagle field in the TCP options structure, which is set to the number two. TCP_CORK is mutually exclusive with the TCP_NODELAY option

TCP_DEFER_ACCEPT: The application caller may sleep until data arrives at the socket, at which time it is awakened. The socket is also awakened when it times out. The caller specifies the number of seconds to wait for data to arrive. This option is unique to Linux, and application

code using it will not be portable to other OSs. The option value is converted to the number of ticks and is kept in the defer_accept field of the TCP option structure.

TCP_INFO: The caller using this option can retrieve lots of configuration information about the socket. This is a Linux-unique option, and code using it will not necessarily be portable to other OSs. The information is returned in the tcp_info structure, defined in file *linux/include/linux/tcp.h*.

```
struct tcp_info
{
```

The first field, tcpi,_state, contains the current TCP state for the connection. The other fields in this structure contain statistics about the TCP connection.

```
    __u8    tcpi_state;
    __u8    tcpi_ca_state;
    __u8    tcpi_retransmits;
    __u8    tcpi_probes;
    __u8    tcpi_backoff;
    __u8    tcpi_options;
    __u8    tcpi_snd_wscale : 4, tcpi_rcv_wscale : 4;

    __u32   tcpi_rto;
    __u32   tcpi_ato;
    __u32   tcpi_snd_mss;
    __u32   tcpi_rcv_mss;

    __u32   tcpi_unacked;
    __u32   tcpi_sacked;
    __u32   tcpi_lost;
    __u32   tcpi_retrans;
    __u32   tcpi_fackets;
```

The following four fields are event time stamps; however, we don't actually remember when an ack was sent in all circumstances.

```
    __u32   tcpi_last_data_sent;
    __u32   tcpi_last_ack_sent;
    __u32   tcpi_last_data_recv;
    __u32   tcpi_last_ack_recv;
```

The last fields are TCP metrics, such as negotiated MTU, send threshold, round-trip time, and congestion window.

```
    __u32   tcpi_pmtu;
    __u32   tcpi_rcv_ssthresh;
    __u32   tcpi_rtt;
    __u32   tcpi_rttvar;
    __u32   tcpi_snd_ssthresh;
    __u32   tcpi_snd_cwnd;
    __u32   tcpi_advmss;
    __u32   tcpi_reordering;
} ;
```

TCP_KEEPCNT: By using this option, the caller can set the number of keepalive probes that TCP will send for this socket before dropping the connection. This option is unique to Linux and should not be used in portable code. The field keepalive_probes in the tcp_opt structure is set to the value of this option. For this option to be effective, the socket level option SO_KEEPALIVE must also be set.

TCP_KEEPIDLE: With this option, the caller may specify the number of seconds that the connection will stay idle before TCP starts to send keepalive probe packets. This option is only effective if the socket option SO_KEEPALIVE is also set for this socket. This is also a nonportable Linux option. The value of this option is stored in the keepalive_time field in the TCP options structure. The value is normally set to a default of two hours.

TCP_KEEPINTVL: This option, also a nonportable Linux option, is used to specify the number of seconds between transmissions of keepalive probes. The value of this option is stored in the keepalive_intvl field in the TCP options structure and is initialized to a value of 75 seconds.

TCP_LINGER2: This option may be set to specify how long an orphaned socket in the FIN_WAIT2 state should be kept alive. The option is unique to Linux and therefore is not portable. If the value is set to zero, the option is turned off and Linux uses normal processing for the **FIN_WAIT_2** and **TIME_WAIT** states. One aspect of this option is not documented anywhere; if the value is less than zero, the socket proceeds immediately to the **CLOSED** state from the **FIN_WAIT_2** state without passing through the **TIME_WAIT** state. The value associated with this option is kept in the linger2 of the tcp_opt structure. The default value is determined by the sysctl, tcp_fin_timeout.

TCP_MAXSEG: This option specifies the maximum segment size set for a TCP socket before the connection is established. The advertised MSS value sent to the peer is determined by this option but won't exceed the interface's MTU. The two TCP peers for this connection may renegotiate the segment size. See Section 8.4.1 for more details on how MSS is used by tcp_sendmsg.

TCP_NODELAY: When set, this option disables the Nagle algorithm. The value is stored in the nonagle field of the tcp_opt structure. This option may not be used if the option TCP_CORK is set. When TCP_NODELAY is set, TCP will send out data as soon as possible without waiting for enough data to fill a segment.

TCP_QUICKACK: This option may be used to turn off delayed acknowledgment by setting the value to one, or enable delayed acknowledgment by setting to a zero. Delayed acknowledgment is the normal mode of operation for Linux TCP. With delayed acknowledgment, ACKs are delayed until they can be combined with a segment waiting to be sent in the reverse direction. If the value of this option is one, the pingpong field in the ack part of tcp_opt is set to zero, which disables delayed acknowledgment. The TCP_QUICKACK option only temporarily affects the behavior of the TCP protocol. If delayed acknowledgment mode is disabled, it could eventually be "automatically" re-enabled depending on the acknowledgment timeout processing and other factors.

TCP_SYNCNT: The caller may use this option to specify the number of SYN retransmits that should be sent before aborting an attempt to establish a connection. This option is unique to Linux and should not be used for portable code. The value is stored in the syn_retries field of the tcp_opt structure.

TCP_WINDOW_CLAMP: By setting this option, the caller may specify the maximum advertised window size for this socket. The minimum allowed for the advertised window is the value SOCK_MIN_RCVBUF divided by two, which is 128 bytes. The value of this option is held in the window_clamp field of tcp_opt for this socket.

8.3 Transport Layer Socket Initialization

In this section, we cover how the transport protocols are initialized when a socket is created. The proto structure contains the mapping from the socket layer generic functions to the protocol specific functions. See Section 8.2.4 for more details about the proto structure. Chapter 5 discussed how an AF_INET socket of type SOCK_STREAM is created, and how the function pointed to by the init field is executed by the inet_create function. When a socket of type SOCK_DGRAM is created, the proto structure is filled in with specific values for UDP. UDP is relatively simple and provides only a datagram service that does not need any internal state information, so it does not require any protocol-specific socket initialization. The proto structure for UDP does not map any function to the init field. Therefore, no UDP specific socket initialization is done at socket creation time. However, for TCP, a SOCK_STREAM socket, the init field of this structure is set to point to the function tcp_v4_init_sock at socket initialization time.

8.3.1 TCP Socket Initialization

In this section, we discuss tcp_v4_init_sock, defined in file *linux/net/ipv4/tcp_ipv4.c* to see how it completes initialization of the SOCK_STREAM type or TCP protocol. Since tcp_v4_init_sock function is called after the sock structure is created, the sock structure has many fields that are already initialized with the value zero and require no further initialization. For details about the sock structure, see Chapter 5. Many of the values initialized by this function are fields in the TCP options structure discussed elsewhere in this chapter.

```
static int tcp_v4_init_sock(struct sock *sk)
{
```

As in most other functions in this chapter, we must obtain a pointer to the TCP options structure.

```
    struct tcp_opt *tp = tcp_sk(sk);
```

The out_of_order_queue is initialized. Unlike the other queues, this queue is unique to TCP and therefore has not been initialized by the socket layer. The transmit timers are initialized by calling tcp_init_xmit_timers. Refer to the section on TCP timers in this chapter for more information.

```
    skb_queue_head_init(&tp->out_of_order_queue);
    tcp_init_xmit_timers(sk);
```

```
tcp_prequeue_init(tp);
```

The retransmit time, rto, and the medium deviation, mdev, which is for Round Trip Time (RTT) measurement, are set to a value of three seconds.

```
tp->rto  = TCP_TIMEOUT_INIT;
tp->mdev = TCP_TIMEOUT_INIT;
```

The send congestion window, cwnd, is initialized to two and it seems strange that it is not zero, but the source code includes the following comment: "So many TCP implementations out there (incorrectly) count the initial SYN frame in their delayed-ACK and congestion control algorithms that we must have the following Band-Aid to talk efficiently to them.—DaveM"

```
tp->snd_cwnd = 2;
```

The send slow start threshold, snd_ssthresh is set to the maximum 32 bit number—effectively disabling the slow start algorithm. The send congestion window clamp, snd_cwnd_clamp, is set to the maximum 16-bit value. The field mss_cache is the minimum segment size for TCP and is initialized to 536 as required [RFC 794].

```
tp->snd_ssthresh = 0x7fffffff;
tp->snd_cwnd_clamp = ~0;
tp->mss_cache = 536;
```

The reordering field of the TCP options structure is initialized to its configured system control value. The socket state, kept in the state field in the sock structure, is initialized to the closed state.

```
tp->reordering = sysctl_tcp_reordering;
sk->state = TCP_CLOSE;
```

The write_space field of the sock structure, sk, is a pointer to a callback function, which is called when buffers are available in the socket's write queue. It is initialized to point to the function tcp_write_space. The use_write_queue field of the sock structure is set to one to indicate that this protocol, (which is TCP of course) uses the socket's write queue.

```
sk->write_space = tcp_write_space;
sk->use_write_queue = 1;
```

The af_specific field of the TCP options structure is set to a set of AF_INET specific functions used by the TCP protocol, which is covered in the next section.

```
sk->tp_pinfo.af_tcp.af_specific = &ipv4_specific;
```

The sndbuf and rcvbuf fields of the sock structure hold the socket options SO_SNDBUF and SO_RCVBUF, which determine the size of the socket's send and receive buffers, respectively. They are initialized to the system control values here as the socket is being initialized, but setsockopt may change them later. Tcp_sockets_allocated is a global defined in tcp.c that holds the number of open TCP sockets.

```
    sk->sndbuf = sysctl_tcp_wmem[1];
    sk->rcvbuf = sysctl_tcp_rmem[1];
    atomic_inc(&tcp_sockets_allocated);
    return 0;
}
```

8.3.2 The Tcp_func Structure for TCP

As we saw previously, TCP socket initialization includes setting the af_specific pointer in the TCP options part of the sock structure to ipv4_specific, which is a pointer to an instance of a tcp_func structure. The tcp_func structure, defined in file *linux/include/linux/tcp.h*, contains a set of IPv4-specific functions for TCP that are dependent on the AF_INET address family. Its purpose is to facilitate port sharing between IPv4 and IPv6. Table 8.3 shows the mapping between these fields and the specific values for TCP, which are initialized as ipv4_specific in *file linux/net/ipv4/tcp_ipv4.c*.

Table 8.3: IPv4 Specific Values for Tcp_func

Tcp_func Field	TCP Actual Value	Description
queue_xmit	ip_queue_xmit	IPv4 network layer transmit function.
send_check	tcp_v4_send_check	Function to calculate IPv4 TCP checksum.
rebuild_header	tcp_v4_rebuild_header	Obtains IPv4 destination address for IP header.
conn_request	tcp_v4_conn_request	Function to process incoming connection request for IPv4.
syn_recv_sock	tcp_v4_syn_recv_sock	Function to create new child socket after receiving SYNACK from peer.
remember_stamp	tcp_v4_remember_stamp	This is a function to save the last received timestamp from a particular peer, used for duplicate segment detection.
net_header_len	sizeof(struct iphdr)	Size of network header. Set to IPv4 header length.
setsockopt	ip_setsockopt	These are the network layer socket option functions for IPv4. They handle IP socket options left over after socket layer and transport layer socket options have been processed.
getsockopt	ip_getsockopt	See setsockopt field above.
addr2sockaddr	v4_addr2sockaddr	Function to generate sockaddr_in type for IPv4.
sockaddr_len	sizeof(struct sockaddr_in)	Size of sockaddr_in for IPv4.

```
struct tcp_func {
    int           (*queue_xmit)     (struct sk_buff *skb);
    void          (*send_check)     (struct sock *sk,
                                     struct tcphdr *th,
                                     int len,
```

```
                                    struct sk_buff *skb);
    int           (*rebuild_header) (struct sock *sk);
    int           (*conn_request)   (struct sock *sk,
                                     struct sk_buff *skb);
    struct sock * (*syn_recv_sock)  (struct sock *sk,
                                     struct sk_buff *skb,
                                    struct open_request *req,
                                    struct dst_entry *dst);

    int           (*remember_stamp) (struct sock *sk);
    __u16         net_header_len;
    int           (*setsockopt)     (struct sock *sk,
                                    int level,
                                    int optname,
                                    char *optval,
                                    int optlen);
    int           (*getsockopt)     (struct sock *sk,
                                    int level,
                                    int optname,
                                    char *optval,
                                    int *optlen);
    void          (*addr2sockaddr)  (struct sock *sk,
                                    struct sockaddr *);
    int           sockaddr_len;
} ;
```

8.4 Initiating a Connection

As we are aware, UDP is a protocol for sending individual datagrams and doesn't actually maintain connections between peer hosts. There is no state information maintained between subsequent UDP packet transmissions. In contrast, TCP is connection oriented, and much of the processing associated with TCP is related to the setup and breakdown of connections. Connections are initiated at the client side by calling the connect socket call. As Table 8.2 shows, when connect is called from the application code, the tcp_v4_connect function is executed for TCP.

Even though UDP doesn't support connections, the connect socket call is supported for UDP. The connect call is supported for datagram sockets, so subsequent send calls can omit the destination address. In this section, we discuss the how the connect socket call is processed for both TCP and UDP.

8.4.1 The connect Call and UDP

The connect call can be made for a UDP socket. When connect is called by the user, the destination address is specified. The main purpose of connect for UDP is to establish the route to the destination and enter it in the routing cache. Once a route is established, subsequent packet transmissions through the UDP socket can use the cached route information. This is called the *fast path* for connected socket. When connect is called on an open SOCK_DGRAM type socket, the function udp_connect, in file *linux/net/ipv4/udp.c*, is called by the socket layer. Sk is a pointer to the sock structure for the open socket, and uaddr is the destination address to which we want to create a route.

```
int udp_connect(struct sock *sk, struct sockaddr *uaddr, int addr_len)
{
```

IPv4-specific address and option information is in the inet_opt structure.

```
    struct inet_opt *inet = inet_sk(sk);
    struct sockaddr_in *usin = (struct sockaddr_in *) uaddr;
```

Rt is a pointer to a route cache entry. See Chapter 9, "The Network Layer, IP," for more information about the route cache. Oif is the index of the output network interface that will carry the packet using the route to the destination.

```
    struct rtable *rt;
    u32 saddr;
    int oif;
    int err;
```

First, we make sure that the specified address is in the correct format for an Internet address.

```
    if (addr_len < sizeof(*usin))
          return -EINVAL;

    if (usin->sin_family != AF_INET)
          return -EAFNOSUPPORT;
```

We call sk_dst_reset to free any old destination cache entry pointed to by the dst field in the sock structure, sk.

```
sk_dst_reset(sk);
```

Since this is may be a bound socket, the network interface may already be known. If so, oif will be index of the outgoing network interface, and if not, it will be zero. Next, we check to see if the destination address in usin is a multicast address. If it is multicast, it doesn't mean that the user is trying to connect to a multicast address; instead, it means that the user will be sending subsequent packets to the same address. In addition, if the destination address is multicast, we get the output interface and source address from the inet_opt structure in the sock, sk.

```
    oif = sk->bound_dev_if;
    saddr = sk->saddr;
    if (MULTICAST(usin->sin_addr.s_addr)) {
        if (!oif)
            oif = sk->protinfo.af_inet.mc_index;
        if (!saddr)
            saddr = sk->protinfo.af_inet.mc_addr;
    }
```

This is the most important call in this function. Ip_route_connect gets a route to the destination address in usin and sets rt to point to the new cached route. If it returns a nonzero value, it wasn't able to find a route or add a new one to the routing cache, so we return the error. If it found a broadcast route, we return an error.

```
    err = ip_route_connect(&rt, usin->sin_addr.s_addr, saddr,
                           RT_CONN_FLAGS(sk), oif, IPPROTO_UDP, inet->sport,
                            usin->sin_port, sk);
    if (err)
        return err;
    if ((rt->rt_flags&RTCF_BROADCAST) && !sock_flag(sk, SOCK_BROADCAST)) {
        ip_rt_put(rt);
        return -EACCES;
    }
    if(!sk->saddr)
```

Here we update the source address and destination address for outgoing packets from the fields in the route cache entry, rt. The destination port is specified by the user.

```
          sk->saddr = rt->rt_src;
    if(!sk->rcv_saddr)
        sk->rcv_saddr = rt->rt_src;
    sk->daddr = rt->rt_dst;
    sk->dport = usin->sin_port;
```

We set the socket state to TCP_ESTABLISHED to indicate that there is a cached route. This value is misleading for a UDP socket, but it is merely to show that the socket has a cached route associated with it. Later, when the user tries to transmit a packet and the source address is missing in the send call, we consider it OK because the state indicates that there is a route established.

```
    sk->state = TCP_ESTABLISHED;
    inet->id = jiffies;
```

Finally, a pointer to the route cache entry is placed in the destination field, dst, of the socket.

```
    sk_dst_set(sk, &rt->u.dst);
    return(0);
}
```

8.4.2 Tcp_v4_connect—Requesting a TCP Connection

Applications using SOCK_STREAM type sockets are classified as either clients or servers. The client requests connections with a server and the server responds to connection requests. The socket call provided to the client to request connections is connect. When connect is executed on an open socket, the socket layer calls a function in the protocol to process the connection request. For SOCK_STREAM type protocols in the AF_INET address family, the function called by the socket layer is tcp_v4_connect defined in file *linux/net/ipv4/tcp_ipv4.c*.

```
int tcp_v4_connect(struct sock *sk, struct sockaddr *uaddr, int addr_len)
{
    struct inet_opt *inet = inet_sk(sk);
```

Tp points to the TCP options, tcp_opt, in the sock structure. The TCP options are discussed in Section 8.7.2. Rt is a route table cache entry. Later, it will point to the cached route for packets being sent through this socket.

```
struct tcp_opt *tp = &(sk->tp_pinfo.af_tcp);
struct sockaddr_in *usin = (struct sockaddr_in *) uaddr;
struct rtable *rt;
```

Daddr is the IP destination address, and nexthop is the IP address of the gateway router for the route if there is one.

```
u32 daddr, nexthop;
int tmp;
int err;
```

After making sure the specified address is in the correct format, we initialize nexthop and the destination address to point the user-specified destination address. Later, nexthop may be changed depending on the route.

```
if (addr_len < sizeof(struct sockaddr_in))
    return(-EINVAL);
if (usin->sin_family != AF_INET)
    return(-EAFNOSUPPORT);

nexthop = daddr = usin->sin_addr.s_addr;
```

This is a check to see if the source routing option is set for this socket.

```
if (inet->opt && inet->opt->srr) {
    if (daddr == 0)
        return -EINVAL;
    nexthop = inet->opt->faddr;
}
```

Now we attempt to get a route to the destination address given the destination address, which for now is in nexthop, the source address, and the network interface. Ip_route_connect will return a zero if successful. It will set rt to point to the new cached route if it found one.

```
if (inet->opt && inet->opt->srr) {
    if (daddr == 0)
        return -EINVAL;
    nexthop = inet->opt->faddr;
}
```

We don’t allow TCP to connect to nonunicast addresses; therefore, if the route in the route cache is not a unicast route, we return an error.

```
if (rt->rt_flags&(RTCF_MULTICAST|RTCF_BROADCAST)) {
    ip_rt_put(rt);
    return -ENETUNREACH;
}
if (!inet->opt || !inet->opt->srr)
    daddr = rt->rt_dst;

if (!inet->saddr)
    inet->saddr = rt->rt_src;
inet->rcv_saddr = inet->saddr;
```

```
    if (tp->ts_recent_stamp && inet->daddr != daddr) {
```

Here, we reset the inherited state.

```
/* Reset inherited state */
        tp->ts_recent    = 0;
        tp->ts_recent_stamp = 0;
        tp->write_seq    = 0;
    }
    if (sysctl_tcp_tw_recycle &&
        !tp->ts_recent_stamp && rt->rt_dst == daddr) {
        struct inet_peer *peer = rt_get_peer(rt);


        if (peer && peer->tcp_ts_stamp + TCP_PAWS_MSL >= xtime.tv_sec) {
            tp->ts_recent_stamp = peer->tcp_ts_stamp;
            tp->ts_recent = peer->tcp_ts;
        }
    }
```

Refer to Chapter 10," Receiving Data in the Transport Layer, UDP and TCP," for more information about the **TIME_WAIT** state. The comments in the code state that this timestamp saving idea comes from an idea of Van Jacobson. The last used timestamp values are saved in the inet_peer structure associated with the route entry when the socket is in the **TIME_WAIT** state. Then, when a new connection is requested, the recent received timestamp, and the timestamp fields in the *tcp_opt* structure, are restored from the saved values in the inet_peer structure. This is a method for detecting duplicate segments once the connection is re-established.

```
    inet->dport = usin->sin_port;
    inet->daddr = daddr;

    tp->ext_header_len = 0;
    if (inet->opt)
        tp->ext_header_len = inet->opt->optlen;
```

This is the minimum allowed MSS value.

```
    tp->mss_clamp = 536;
```

Although, we are in the process of doing an active open, the identity of the socket is not known. We don't know the source port, sport, because we have not yet assigned the ephemeral port. However, we put ourselves in the **TCP_SYN_SENT** state and enter sk into the TCP connection's hash table. The remainder of the work for initialization the connection will be finished later.

```
tcp_set_state(sk, TCP_SYN_SENT);
```

We call tcp_v4_hash_connect to enter the socket in the connect hash table. This assigns an ephemeral port for the connection, and puts it in the sport field of the sock. The ephemeral port number serves as the hash code for finding the connected socket in the hash table. Later, when

the socket is in **ESTABLISHED** state and incoming packets arrive for the port, TCP can quickly find the correct open socket.

```
    err = tcp_v4_hash_connect(sk);
    if (err)
        goto failure;
    err = ip_route_newports(&rt, inet->sport, inet->dport, sk);
    if (err)
        goto failure;
```

We commit the destination for this connection by setting the dst field of the sock structure, sk, to point to the destination associated with the route. When a route is for a packet sent to an external destination, dst will point to the gateway. However, if the destination is for a directly connect host, dst will point to information about the host.

```
    __sk_dst_set(sk, &rt->u.dst);
    tcp_v4_setup_caps(sk, &rt->u.dst);
    tp->ext2_header_len = rt->u.dst.header_len;
```

We initialize the first sequence number.

```
    if (!tp->write_seq)
        tp->write_seq = secure_tcp_sequence_number(inet->saddr,

                                                   inet->daddr,
                                                inet->sport,
                                                usin->sin_port);

        inet->id = tp->write_seq ^ jiffies;
```

We call tcp_connect to complete the work of setting up the connection including transmitting the **SYN**.

```
    err = tcp_connect(sk);
    rt = NULL;
    if (err)
        goto failure;
    return 0;
failure:
```

If we failed, we take the socket out of the hash table for connected sockets and release the local port.

```
    tcp_set_state(sk, TCP_CLOSE);
    ip_rt_put(rt);
    sk->sk_route_caps = 0;
    inet->dport = 0;
    return err;
}
```

8.5 Sending Data From a Socket via UDP

In this section, we show how the UDP protocol processes a request from the application layer to send a datagram. When a user program writes data to a SOCK_DGRAM type socket by calling sendmsg or any other application layer sending function, the UDP protocol will process the request. UDP is fairly simple, and mostly consists of a framing layer. It doesn't do much more than pre-pend the user data with the UDP header. Unlike TCP, there is no buffering or connection management. To construct the UDP header, the destination and source port are required. Usually, the destination port and IP address are specified together in the peer socket's name. Generally, when the port is known, the IP address is known, too. If the destination IP address is specified in the socket name, it is saved before the packet is passed to IP. In addition, before passing the packet on to IP, UDP checks to if the destination is internal or whether IP must determine a route for the packet. Moreover, the destination address may be a multicast or broadcast address, in which case IP does not need to route the packet. Before UDP is done, the source and destination ports are placed in the UDP header and the IP address is kept for later processing by IP.

It is interesting to note how the Linux TCP/IP stack is built for efficiency. All efforts have been taken to avoid copying or unnecessary processing. Almost all the complexity of processing is outside the data path of the packet. There is only one copy of the actual data and only one level of processing in UDP.

When the application calls one of the write functions on an open socket, the socket layer calls the function pointed to by the sendmsg field in the prot structure. See Chapter 5, "Linux Sockets," for detailed information about what happens at the socket layer. This results in a call to udp_sendmsg, which is found in file *linux/net/ipv4/udp.c*. This is the sending function that is executed for SOCK_DGRAM type sockets. See Table 8.1 for a list of all the protocol block functions for UDP.

```
int udp_sendmsg(struct sock *sk, struct msghdr *msg, int len)
{
```

We retrieve the protocol specific parts of the sock structure.

```
    struct inet_opt *inet = inet_sk(sk);
    struct udp_opt *up = udp_sk(sk);
```

Ipc will hold the return from the function ip_cmsg_send discussed in Section 8.3.1.

```
    int ulen = len;
    struct ipcm_cookie ipc;
    struct rtable *rt = NULL;
    int free = 0;
    int connected = 0;
```

Daddr is for the destination IP address, and tos is for the TOS field of the IP header.

```
    u32 daddr, faddr, saddr;
    u16 dport;
```

```
    u8  tos;
    int err;
    int corkreq = up->corkflag || msg->msg_flags&MSG_MORE;
```

The first thing udp_sendmsg does is check for an out-of-range value in the len field, and checks to see if the caller has requested any illegal flags for this type of socket. The only illegal flag for UDP is MSG_OOB, which is used only for SOCK_STREAM type sockets.

```
    if (len < 0 || len > 0xFFFF)
        return -EMSGSIZE;
    if (msg->msg_flags&MSG_OOB)
        return -EOPNOTSUPP;
```

Now, check to see if there are any pending frames. The socket lock must be held while the socket is corked for processing the pending frames.

```
    if (up->pending) {
        lock_sock(sk);
        if (likely(up->pending)) {
            if (unlikely(up->pending != AF_INET)) {
                release_sock(sk);
                return -EINVAL;
            }
            goto do_append_data;
        }
        release_sock(sk);
    }
```

We add the UDP header length to the length of the user data.

```
    ulen += sizeof(struct udphdr);
```

Next, udp_sendmsg checks to make sure that either there is a valid destination address specified or the socket is in a connected state. It verifies the address by checking the *name* field in the msghdr structure. If the application caller invoked the sendto socket API call, then the name field of the msghdr structure will contain the address originally specified in the to field of the sendto call.

```
    if (msg->msg_name) {
        struct sockaddr_in * usin = (struct sockaddr_in*)msg->msg_name;
        if (msg->msg_namelen < sizeof(*usin))
            return -EINVAL;
        if (usin->sin_family != AF_INET) {
            if (usin->sin_family != AF_UNSPEC)
                return -EINVAL;
        }
```

At this point, we get the destination and source addresses. The port is placed in the dest field of the udphdr structure pointed to by uh in the fakehdr structure.

```
        daddr = usin->sin_addr.s_addr;
        dport = usin->sin_port;
        if (dport == 0)
```

```
            return -EINVAL;
    }  else {
```

We know that the destination address specified was NULL. However, we can allow the packet to be transmitted if the socket is a connected UDP socket, which is indicated when the state field is set to TCP_ESTABLISHED. If the socket is connected, we assume that the destination address is already known.

```
        if (sk->state != TCP_ESTABLISHED)
            return -EDESTADDRREQ;
        daddr = inet->daddr;
        dport = inet->dport;
```

If the socket is connected, then routing in the IP layer can use the “fast path” to bypass a routing table lookup and use the destination cache entry directly.

```
        connected = 1;
      }
. . .
```

The Udphdr structure, defined in file *linux/include/linux/udp.h*, contains the actual UDP header including the source port, destination port, length, and checksum. In the udp_sendmsg function, the UDP header is a field in ufh, the UDP fake header structure.

```
struct udphdr {
    __u16   source;
    __u16   dest;
    __u16   len;
    __u16   check;
} ;
```

If no valid destination address is found, udp_sendmsg returns EINVAL.

8.5.1 Handling Control Messages

Let’s continue with the udp_sendmsg function to see how it handles the control messages. The next thing udp_sendmsg does is determine if the argument msg points to a control message. It checks the field, msg_controllen, for nonzero value. The structure ipcm_cookie, pointed to by ipc, holds the result of the control message processing. Initially, some fields of ipc are initialized, such as the IP options and the interface (if there is one bound to this socket).

```
. . .
    ipc.addr = inet->saddr;
    ipc.oif = sk->bound_dev_if;
    if (msg->msg_controllen) {
```

The control messages are also called *ancillary data* and are part of the IPv6 sockets interface specification. For more information on control messages, see the man page, cmsg(3), and "Advanced Sockets API for IPv6," [RFC 2292]. If msg contains a pointer to a control message, the function ip_cmsg_send, defined in file *linux/net/ipv4/ip_sockglue.c*, processes the request. These control messages are a way of setting and retrieving UDP information such as the address

and port. Fields in the inet options structure, which contains the addressing information, can be retrieved directly by the control message.

```
        err = ip_cmsg_send(msg, &ipc);
        if (err)
            return err;
        if (ipc.opt)
            free = 1;
        connected = 0;
    }
    if (!ipc.opt)
        ipc.opt = inet->opt;
    saddr = ipc.addr;
    ipc.addr = faddr = daddr;
    if (ipc.opt && ipc.opt->srr) {
        if (!daddr)
            return -EINVAL;
        faddr = ipc.opt->faddr;
        connected = 0;
    }
. . .
```

Ip_cmsg_send returns the results in ipc, which points to a structure called ipcm_cookie, defined in *linux/include/linux/ip.h*.

```
    struct ipcm_cookie
    {
        u32                     addr;
        int                     oif;
        struct ip_options       *opt;
    } ;
```

Two types of control messages are processed by UDP. I_RETOPTS retrieves the options field from the IP header and returns a pointer to the options in the *opt* field of ipc. If IP_PKTINFO was specified in the control message, ip_cmsg_send returns the interface index in the oif field of ipc and the interface's IP address in the addr field.

8.5.2 Passing the Packet to IP Output

We continue with our examination of the udp_msgsend function. At this point, most of the information needed for the UDP header is established. Now, we will check to see how this packet is supposed to be routed.

```
. . .
    tos = RT_TOS(sk->protinfo.af_inet.tos);
    if (sk->localroute ||
        (msg->msg_flags&MSG_DONTROUTE) ||
        (ipc.opt && ipc.opt->is_strictroute)) {
        tos |= RTO_ONLINK;
        connected = 0;
    }
```

In addition, if the packet is for transmission through a connected socket, it is assumed that the destination is already known and there is already a route to the destination address in this packet. If the destination address is a multicast address, there is no need to route the packet either.

```
if (MULTICAST(daddr)) {
    if (!ipc.oif)
        ipc.oif = inet->mc_index;
    if (!saddr)
        saddr = inet->mc_addr;
    connected = 0;
}
if (connected)
    rt = (struct rtable*)sk_dst_check(sk, 0);
if (rt == NULL) {
```

If we don't have a route, we prepare for a search of the routing table by building an flow information structure.

```
        struct flowi fl = {  .oif = ipc.oif,
                           .nl_u = {  .ip4_u =
                            {  .daddr = faddr,
                              .saddr = saddr,
                              .tos = tos }  } ,
                              .proto = IPPROTO_UDP,
                              .uli_u = {  .ports =
                                 {  .sport = inet->sport,
                                   .dport = dport }  }  } ;
```

We call ip_route_output to try to come up with a route. See Chapter 9 for more information about ip_route_output and how the routing table entries are manipulated and searched.

```
        err = ip_route_output_flow(&rt, &fl, sk,
                                   !(msg->msg_flags&MSG_DONTWAIT));
        if (err)
            goto out;
        err = -EACCES;
```

If the routing cache entry indicates broadcast but the socket did not have the SO_BROADCAST flag set, then it is an error.

```
        if (rt->rt_flags&RTCF_BROADCAST &&
            !sock_flag(sk, SOCK_BROADCAST))
            goto out;
```

Now that there is a route, if the socket is connected, we set a pointer to the route in the destination cache. See Chapter 6 for more information about the destination cache.

```
        if (connected)
            sk_dst_set(sk, dst_clone(&rt->u.dst));
    }
```

Udp_msgsend checks if the application requested routing confirmation by looking for the flag MSG_CONFIRM in the flags field of the msghdr structure. If required, dst_confirm is called to confirm the route.

```
    if (msg->msg_flags&MSG_CONFIRM)
        goto do_confirm;
```

At this point, ip_build_xmit is called to pass the packet on to the IP layer.

```
back_from_confirm:
    saddr = rt->rt_src;
    if (!ipc.addr)
        daddr = ipc.addr = rt->rt_dst;
    lock_sock(sk);
    if (unlikely(up->pending)) {
```

Check that the sock is already corked. This must be a cork application bug.

```
        release_sock(sk);

        NETDEBUG(if (net_ratelimit()) printk(KERN_DEBUG "udp cork app
bug 2\ n"));
        err = -EINVAL;
        goto out;
    }
```

Cork the socket in order to append additional data.

```
    inet->cork.fl.fl4_dst = daddr;
    inet->cork.fl.fl_ip_dport = dport;
    inet->cork.fl.fl4_src = saddr;
    inet->cork.fl.fl_ip_sport = inet->sport;
    up->pending = AF_INET;
```

Now we send the data to IP by calling ip_append_data, which builds a large datagram from individual pieces of data. The second argument, ip_generic_getfrag, is a callback function executed by IPv4 when it is ready to copy the actual data from user space to the datagram.

```
do_append_data:
    up->len += ulen;
    err = ip_append_data(sk,ip_generic_getfrag,msg->msg_iov, ulen,
                         sizeof(struct udphdr), &ipc, rt,
                          corkreq ? msg->msg_flags|MSG_MORE : msg->msg_flags);
    if (err)
        udp_flush_pending_frames(sk);
    else if (!corkreq)
        err = udp_push_pending_frames(sk, up);
    release_sock(sk);
out:
    ip_rt_put(rt);
    if (free)
        kfree(ipc.opt);
    if (!err) {
        UDP_INC_STATS_USER(UdpOutDatagrams);
```

```
        return len;
    }
    return err;
do_confirm:
```

If a neighbor cache entry exists for this route, dst_confirm timestamps an existing entry in the neighbor cache.

```
    dst_confirm(&rt->u.dst);
```

If the MSG_PROBE flag was set for this socket, the transmit path should be probed for maximum MTU but the packet should not be sent.

```
    if (!(msg->msg_flags&MSG_PROBE) || len)
        goto back_from_confirm;
    err = 0;
    goto out;
}
```

8.5.3 Copying Data from User Space to a Datagram

This section covers the function that gets pieces of data from user space to append to a datagram, ip_generic_getfrag. This function is defined in *linux/net/ipv4/dp_output.c*. The function is called as a callback from ip_append_data function, also in the same file.

ip_append_data is executed whether or not checksums have been disabled. The user data is organized in one or more buffer pointers referenced by iov. Generally, there is only one buffer for each UDP datagram.

```
int ip_generic_getfrag(void *from, char *to, int offset, int len,
int odd, struct sk_buff *skb)
{
```

The UDP header does not need to be copied from user space to kernel space. It has already been built in kernel space by the udp_msgsend function. We call one of two functions to copy the user data depending on whether our output hardware has hardware checksum capability. One of two functions, csum_partial_copy_fromiovecend, calculates the partial checksums while copying. The other function, memcpy_fromiovecend, is the one we use if we will be calculating the checksum later in hardware because this function does not calculate a checksum. A UDP checksum includes parts of the IP header, the source and destination IP addresses, IP header protocol, and length fields. Therefore, the checksum calculation is done in three stages: once for the data, again for the actual UDP header, and then a third time with the fields from the IP header.

```
    struct iovec *iov = from;
    if (skb->ip_summed == CHECKSUM_HW) {
        if (memcpy_fromiovecend(to, iov, offset, len) < 0)
        return -EFAULT;
    }  else {
        unsigned int csum = 0;
```

The data is copied from user space to kernel space by the iovec utility routine. The checksum is calculated on the data while copying for efficiency. It is important to avoid manipulating the packet contents twice. The data must be copied from user space via the iov pointer to a location within a sk_buff in kernel space referenced by the to argument.

```
        if (csum_partial_copy_fromiovecend(to, iov, offset, len, &csum) < 0)
            return -EFAULT;
        skb->csum = csum_block_add(skb->csum, csum, odd);
    }
    return 0;
}
```

8.6 Sending Data From a Socket via TCP

The best way to describe the implementation of a complex protocol like TCP is to follow the data as it flows through the protocol. For the remaining part of this chapter, the send-side processing of TCP will be examined. It is important to keep in mind that TCP is probably the most complex part of the TCP/IP protocol suite. Because TCP provides a connection-oriented service at the transport layer, it is far more complicated than UDP. TCP must manage the relationship between the local host and the remote host, including retransmitting bad and lost packets, managing the connection state machine, buffering the data, and setting up and breaking down the connection. In addition, as will be seen, it does much more than this. Linux TCP implements all of the enhancements for security, reliability, and performance that have been developed in more than 20+ years since TCP was first specified in RFC 793. This section focuses primarily on the connection with the socket, how data is removed from the application and placed in queues for transmission. Next, Section 8.7 discusses the actual packet transmission. Section 8.8 covers the data structures used to keep control variables and configuration options, and Section 8.9 covers the TCP timers used primarily on the send side of the connection. Figure 8.1 shows the TCP connection state for the sending side.

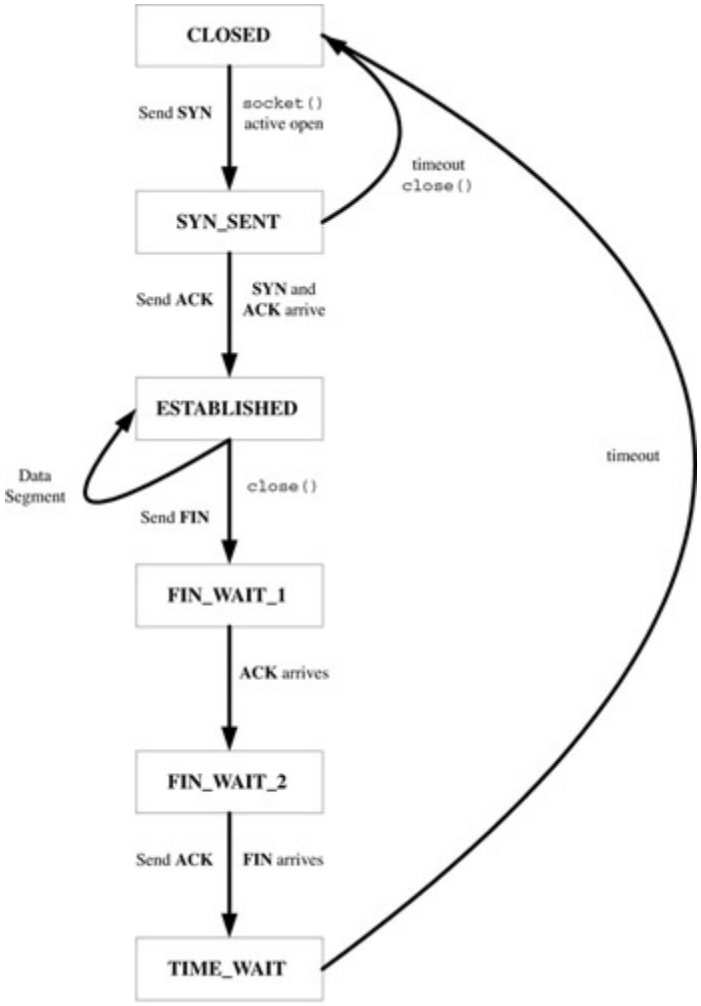

Figure 8.1: TCP send state diagram.

Let's look at how the send side of the protocol handles write information passed from the socket layer. Once an application layer program opens a socket of type SOCK_STREAM, the TCP protocol is invoked to process all write requests for data transfers through the open socket. The transmission of data through TCP is controlled primarily by the state of the connection with the peer and the availability of data in the send buffer. Because of the asynchronous nature of TCP, actual data transmission is independent of the application layer, as it writes data into the socket layer buffers. In general, TCP gathers the data into segments and passes it to the peer machine when there is available bandwidth in the network. This is different from UDP where the user-level process controls transmission, because with UDP, a write of data to the socket results directly in the transmission of a datagram. When any of the socket layer write functions are invoked by the application, TCP copies the data into a list of socket buffers, which are queued up for later transmission. Most of the work associated with writing in a socket involves determining the type of socket buffer and managing the queue of buffers. It must be determined whether the transmission interface has scatter-gather capability, otherwise known as *chained DMA*. If the interface has scatter-gather support, TCP sets up a transfer of a chain of buffers to the networking device. In this case, TCP uses the fragment list in the shared info structure of the skbuf. In addition, TCP maintains a queue of socket buffers, so the send function must also

determine if there is room for more data in the current socket buffer or if a new one must be allocated.

8.6.1 The Tcp_sendmsg Function

The tcp_msgsend function, defined in file *linux/net/ipv4/tcp.c*, is invoked when any user-level write or message sending function is invoked on an open SOCK_STREAM type socket. All the write functions are converted to calls to this function at the socket layer. The best way to show the operation of TCP on the send side is to examine this function, so we will follow it to see how it processes the data.

The socket layer is able to find tcp_sendmsg because it is referenced by the sendmsg field of the protocol block structure, prot, which is initialized at compile time in the file *linux/net/ipv4/tcp_ipv4.c*. The protocol block functions for TCP are shown in Table 8.2. Tcp_sendmsg collects all the TCP header information possible, copies the data into socket buffers, and queues the socket buffers for transmission. It makes heavy use of the TCP options structure described. Tcp_sendmsg also sets many fields in the TCP control block structure, described in Section 8.5.1, which is used to pass TCP header information to the transmission side of the TCP protocol.

```
int tcp_sendmsg(struct sock *sk, struct msghdr *msg, int size)
{
```

The variable iov is to retrieve the IO vector pointer in the message header structure in the argument msg. Tp points to the TCP options that will be retrieved from the sock structure sk. Skb is a pointer to a socket buffer that will be allocated to hold the data to be transmitted. Iovlen is set to the number of elements in the iovec.

```
    struct iovec *iov;
```

The TCP options are retrieved from the sock structure and the sock structure is locked.

```
    struct tcp_opt *tp = tcp_sk(sk);
    struct sk_buff *skb;
    int iovlen, flags;
```

Mss_now holds the current maximum segment size (MSS) for this open socket.

```
    int mss_now;
    int err, copied;
    long timeo;
    lock_sock(sk);
    TCP_CHECK_TIMER(sk);
    flags = msg->msg_flags;
```

The value of the SO_SNDTIMEO option is put in timeo unless the MSG_DONTWAIT flag was set for the socket.

```
    timeo = sock_sndtimeo(sk, flags&MSG_DONTWAIT);
```

The next thing tcp_sendmsg does is wait for a connection to be established before sending the packet. The state of the connection for this open socket is checked, and if the connection is not already in the TCPF_ESTABLISHED or TCPF_CLOSE_WAIT state, TCP is not ready to send data so it must wait for a connection to be established. The value of the timeout set with the SO_SNDTIMEO socket option is passed into the wait_for_tcp_connect function.

```
if ((1 << sk->state) & ~(TCPF_ESTABLISHED | TCPF_CLOSE_WAIT))
    if((err = wait_for_tcp_connect(sk, flags, &timeo)) != 0)
        goto out_err;
```

The SOCK_ASYNC_NOSPACE bit in flags is cleared to indicate that this socket is not currently waiting for more memory.

```
clear_bit(SOCK_ASYNC_NOSPACE, &sk->sk_socket->flags);
```

Mss_now is set to the current mss for this socket. The function tcp_current_mss takes into account MTU discovery, the negotiated MSS, and the TCP_MAXSEG option. See Section 8.5 on TCP utility functions for more details.

```
mss_now = tcp_current_mss(sk);
```

Now, set up to send the data. Get iov and iovlen from the msghdr structure. Get ready to start copying the data into socket buffers.

```
iovlen = msg->msg_iovlen;
iov = msg->msg_iov;
copied = 0;
err = -EPIPE;
```

If the application called shutdown with how set to one, this means that a socket shutdown on the send side of the socket has been requested, so get out.

```
if (sk->err || (sk->shutdown&SEND_SHUTDOWN))
    goto do_error;
```

The IO vector structure is the mechanism where data is retrieved from user space into kernel space. Iovlen is the number of iov buffers queued up by the socket send call.

```
while (—iovlen >= 0) {
```

Seglen is the length of each iov, and from points to the data to be copied.

```
    int seglen=iov->iov_len;
    unsigned char * from=iov->iov_base;
    iov++;
    while (seglen > 0) {
        int copy;
        skb = sk->write_queue.prev;
```

If send_head is null, there are not yet any segments queued for sending. Mss_now is the current negotiated MSS.

```
            if (!tp->send_head == NULL
                (copy = mss_now - skb->len) <= 0) {
```

If send_head is NULL, it is necessary to allocate a new segment. Tcp_alloc_pskb is used as a TCP-specific wrapper for alloc_skb. It allocates a paged socket buffer. Paged socket buffers include the socket buffer shared info and are used for efficient handling of TCP segments and IP fragments. They are particularly suitable for network interfaces that have scatter-gather DMA capability that can send a chain of buffers via DMA with minimum overhead. Tcp_sendmsg calls select_size to round off the requested size to the nearest TCP segment size or page size determined from the MSS. Skb_entail sets up the flags and sequence numbers in the TCP control block and puts the new skb on the write queue. Copy is the amount of data to copy to the segment. It is set to mss_now, the most recent negotiated value of MSS, which is the best indicator of segment size at this point.

```
new_segment:
                if (!tcp_memory_free(sk))
                    goto wait_for_sndbuf;
                skb = tcp_alloc_pskb(sk,
                                     select_size(sk, tp),
                                     0,
                                     sk->allocation);
                if (!skb)
                    goto wait_for_memory;
```

Check to see whether we can use the hardware checksum capability.

```
                if (sk->sk_route_caps &
                    (NETIF_F_IP_CSUM | NETIF_F_NO_CSUM | NETIF_F_HW_CSUM))
                    skb->ip_summed = CHECKSUM_HW;
                skb_entail(sk, tp, skb);
                copy = mss_now;
            }
. . .
```

8.6.2 Copying the Data from User Space to the Socket Buffer

If possible, tcp_sendmsg tries to squeeze the data into the header portion of the skb before allocating a new segment, so it tries to determine the actual address where the data will be copied.

```
. . .
            if (copy > seglen)
                copy = seglen;
            if (skb_tailroom(skb) > 0) {
```

Skb_tailroom returns the amount of room in the end of the skb, and copy is set to the amount of available space if there is any. Skb_add_data copies the data to the skb and calculates the checksum while it is doing the copy. The partial checksum in progress is placed in the csum field of the socket buffer, skb. In Chapter 7, "Linux Memory Allocation and Skbuffs," Table 7.1 includes a description of the socket buffer structure.

```
                if (copy > skb_tailroom(skb))
                    copy = skb_tailroom(skb);
```

```
            if ((err = skb_add_data(skb, from, copy)) != 0)
                goto do_fault;
        }  else {
```

If there was no tailroom in the main part of the socket buffer, skb, we try to find room in the last fragment attached to skb. If there is no room in the fragment, we allocate a new page and attach it by putting a pointer to it at the end of the frags array in the shared info part of the skb. In Chapter 7, Section 7.5.1 has a complete discussion of how the frag array is laid out in the socket buffers. Merge is set to one if there is any room in the last page, and i is set to the number of frags already in the socket buffer. Page is declared to point to a memory-mapped page, and off is set to the offset to the start of the data to be copied from the msghdr structure. TCP_PAGE is used several times in the tcp_sendmsg function. It is a macro that updates the sndmsg_page field in the tcp_options structure to hold a pointer to the current page in the frags array. The macro, TCP_OFF, also defined in *linux/net/ipv4/tcp.c*, updates the sndmsg_off field with an updated offset into the page. Both these macros are defined in *tcp.c*.

```
            int merge = 0;
            int i = skb_shinfo(skb)->nr_frags;
            struct page *page = TCP_PAGE(sk);
            int off = TCP_OFF(sk);
```

Tcp_sendmsg checks to see if there is space in the last page in the socket buffer by calling can_coalesce, which returns a one if there is room. Next, the function determines if a new page is needed or a complete new socket buffer must be allocated. To do this, tcp_sendmsg checks if the frag slots in the skb are full or whether the outgoing interface has scatter-gather capability. SG means that the interface hardware can efficiently exploit socket buffers with attached pages by doing segmented DMA.

```
            if (can_coalesce(skb, i, page, off) && off != PAGE_SIZE) {
                merge = 1;
```

We check to see if there is any reason why we can't add a new fragment.

```
            }  else if (i == MAX_SKB_FRAGS ||
                       (!i && !(sk->route_caps&NETIF_F_SG))) {
```

Perhaps all the page slots are used or the interface is not capable of scatter-gather DMA. In any case, we know that we can't add more data to the current segment. Therefore, we call tcp_mark_push to set the TCPCB_FLAG_PSH flag in the control buffer for this connection. This will mean that the PSH flag will be set in the TCP header of the current skb, and Tcp_sendmsg jumps to the new_segment label to allocate a new skb.

```
                tcp_mark_push(tp, skb);
                goto new_segment;
            }  else if (page) {
```

If the page has been allocated, off is aligned to the page boundary. Then, as an extra error check, off is checked for validity value, and if it isn't valid, the page is freed and removed from the skb.

```
                off = (off+L1_CACHE_BYTES-1)&~(L1_CACHE_BYTES-1);
```

```
        if (off == PAGE_SIZE) {
            put_page(page);
            TCP_PAGE(sk) = page = NULL;
        }
    }
```

A new page is allocated if necessary, and copy, the length of data, is corrected for the page size.

```
    if (!page) {
```

Allocate the new cache page.

```
        if (!(page=tcp_alloc_page(sk)))
            goto wait_for_memory;
        off = 0;
    }
    if (copy > PAGE_SIZE-off)
        copy = PAGE_SIZE-off;
```

Finally, tcp_sendmsg is ready to copy the data from user space to kernel space. It calls tcp_copy_to_page to do the copying, which in turn calls csum_and_copy_from_user to copy the data efficiently by simultaneously calculating a partial checksum. If tcp_copy_to_page returns an error, the allocated but empty page is attached to the sock structure for this open socket, sk, in the sndmsg_page field so the page will be de-allocated when the socket is released.

```
    err = tcp_copy_to_page(sk, from, skb, page, off, copy);
    if (err) {
        if (TCP_PAGE(sk) == NULL) {
            TCP_PAGE(sk) = page;
            TCP_OFF(sk) = 0;
        }
        goto do_error;

    }
```

Now that the copy has been done, the skb is updated to reflect the new data. If merge is nonzero, it means that the data was merged into the last frag, so the size field in that frag must be updated. Otherwise, a new page was allocated, so fill_page_desc is called, which updates the frag array in the skb with information about the new page. The sndmsg_page field in the tcp_options structure is updated to point to the next page, and the sndmsg_off field gets an updated offset into the page.

```
    if (merge) {
        skb_shinfo(skb)->frags[i-1].size += copy;
    }  else {
        fill_page_desc(skb, i, page, off, copy);
        if (TCP_PAGE(sk)) {
            get_page(page);
        }  else if (off + copy < PAGE_SIZE) {
            get_page(page);
            TCP_PAGE(sk) = page;
        }
    }
    TCP_OFF(sk) = off + copy;
```

```
            }
. . .
```

8.6.3 Tcp_sendmsg Completion

At this point, most of the work of tcp_sendmsg is done. The data has been copied from user space to the socket buffer. The socket buffer's frags array has been updated with a list of pages ready for sending segments. A zero value in the variable copied indicates that this is the first time through the while loop for the initial segment so the PSH flag in the TCP header is set to zero. As we will see in this section, the TCP header is not set directly at this point. Instead, the intended values are saved in the TCP control block for later when the queued socket buffers are removed for transmission. Stevens has an excellent discussion in Section 20.5 of the use of the PSH flag in TCP [STEV94].

```
. . .
            if (!copied)
                TCP_SKB_CB(skb)->flags &= ~TCPCB_FLAG_PSH;
```

The write_seq field in the TCP options structure is updated with the amount of data processed in this trip through the while loop. The end_seq field in the TCP control block is also updated with the amount of data that was processed. From, which points to the data source in the msg argument, and copied, which holds the total number of bytes processed so far, are updated for this iteration. Iovlen is the number of iovs in the msghdr structure, and seglen was initialized to this value at the start of the function. Each loop through this code processes one iov.

```
            tp->write_seq += copy;
            TCP_SKB_CB(skb)->end_seq += copy;
            from += copy;
            copied += copy;
            if ((seglen -= copy) == 0 && iovlen == 0)
                goto out;
            if (skb->len != mss_now || (flags&MSG_OOB))
                continue;
```

At this point, we decide if small segments should be pushed out or held. We call forced_push to check if we must send now no matter what the segment size is. If so, the PSH flag is set and the segments are transmitted.

```
            if (forced_push(tp)) {
                tcp_mark_push(tp, skb);
                __tcp_push_pending_frames(sk, tp, mss_now, TCP_NAGLE_PUSH);
            }  else if (skb == tp->send_head)
                tcp_push_one(sk, mss_now);
            continue;
```

This is where tcp_sendmsg ends up if there is insufficient number of buffers or pages. It waits until there are a sufficient number of buffers available.

```
wait_for_sndbuf:
            set_bit(SOCK_NOSPACE, &sk->socket->flags);
wait_for_memory:
            if (copied)
```

```
                tcp_push(sk, tp, flags&~MSG_MORE, mss_now, TCP_NAGLE_PUSH);
            if ((err = wait_for_tcp_memory(sk, &timeo)) != 0)
                goto do_error;
            mss_now = tcp_current_mss(sk);
        }
    }
```

This is the label where the code jumped if it is time to push the remaining data and exit the function. The value of copied is returned, indicating to the application program how much data was transmitted.

```
out:
    if (copied)
        tcp_push(sk, tp, flags, mss_now, tp->nonagle);
    TCP_CHECK_TIMER(sk);
    release_sock(sk);
    return copied;
```

These last three labels mean that we are at the end of tcp_sendmsg. This is where we finally end up if errors were detected in the process of copying and processing the data.

```
do_fault:
    if (skb->len == 0) {
        if (tp->send_head == skb)
            tp->send_head = NULL;
        __skb_unlink(skb, skb->list);
        tcp_free_skb(sk, skb);
    }
do_error:
    if (copied)
        goto out;
out_err:
    err = tcp_error(sk, flags, err);
    TCP_CHECK_TIMER(sk);
    release_sock(sk);
    return err;
}
```

8.7 TCP Output

The previous discussion about TCP in Section 8.4 focused primarily on how the TCP protocol was interfaced to the socket, and how data was removed from the application and placed in queues for transmission. In this section, we cover the actual packet transmission (see Figure 8.2).

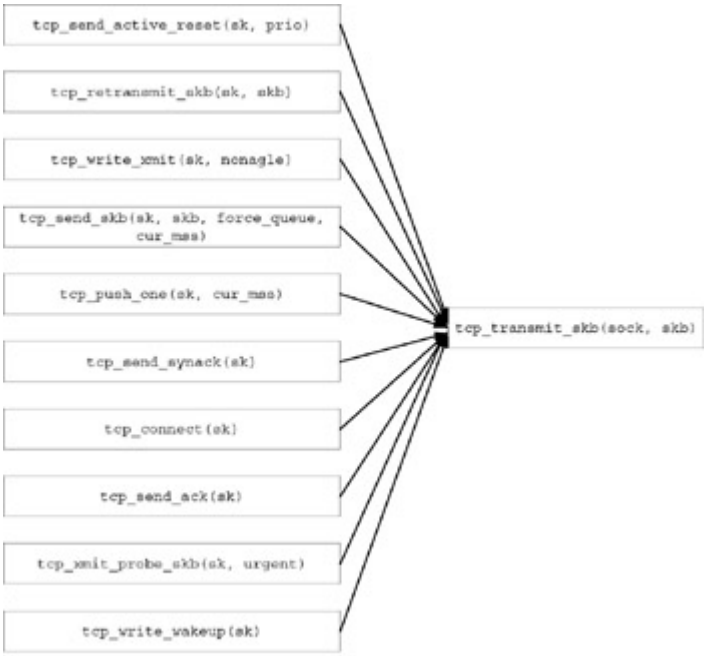

Figure 8.2: TCP transmit sequence.

8.7.1 Transmit the TCP Segments, the Tcp_transmit_Skb Function

Now it is time to transmit the TCP segments, and the function tcp_Transmit_skb does the actual packet transmission. It sends the packets that are queued to the socket. It can be called from anywhere in TCP state processing when there is a request to send a segment. Earlier, we saw how tcp_sendmsg readied the segments for transmission and queued them to the socket's write queue.

In tcp_transmit_skb, we build the TCP packet header and pass the packet on to IP.

```
int tcp_transmit_skb(struct sock *sk, struct sk_buff *skb)
{
```

If we receive a NULL socket buffer, we do nothing.

```
    if(skb != NULL) {
```

Inet points to the inet options structure, and tp points to the TCP options structure. It is in tcp_opt where the socket keeps most of the configuration and connection state information for TCP. Tcb points to the TCP control buffer containing most of the flags as well as the partially constructed TCP header. Th is a pointer to the TCP header. Later, it will point to the header part of the skb, and sysctl_flags is for some critical parameters configured via sysctl and setsockopt calls.

```
        struct inet_opt *inet = inet_sk(sk);
        struct tcp_opt *tp = inet_sk(sk);
        struct tcp_skb_cb *tcb = TCP_SKB_CB(skb);
        int tcp_header_size = tp->tcp_header_len;
        struct tcphdr *th;
        int sysctl_flags;
        int err;
```

```
#define SYSCTL_FLAG_TSTAMPS0x1
#define SYSCTL_FLAG_WSCALE0x2
#define SYSCTL_FLAG_SACK0x4
        sysctl_flags = 0;
```

Here, we check to see if this outgoing packet is a **SYN** packet, and if so, we check for the presence of certain TCP options in the flags field of the control buffer structure. This is because the TCP header length may need to be extended to account for certain TCP options, which generally include timestamps, window scaling, and Selective Acknowledgement (SACK) [RFC 2018].

```
        if (tcb->flags & TCPCB_FLAG_SYN) {
            tcp_header_size = sizeof(struct tcphdr) + TCPOLEN_MSS;
            if(sysctl_tcp_timestamps) {
                tcp_header_size += TCPOLEN_TSTAMP_ALIGNED;
                sysctl_flags |= SYSCTL_FLAG_TSTAMPS;
            }
            if(sysctl_tcp_window_scaling) {
                tcp_header_size += TCPOLEN_WSCALE_ALIGNED;
                sysctl_flags |= SYSCTL_FLAG_WSCALE;
            }
            if(sysctl_tcp_sack) {
                sysctl_flags |= SYSCTL_FLAG_SACK;
                if(!(sysctl_flags & SYSCTL_FLAG_TSTAMPS))
                    tcp_header_size += TCPOLEN_SACKPERM_ALIGNED;
            }
        }  else if (tp->eff_sacks) {
```

The following processing is for the SACK option. If we are sending SACKs in this segment, we increment the header to account for the number of SACK blocks that are being sent along with this packet. The header length is adjusted by eight for each SACK block.

```
            tcp_header_size += (TCPOLEN_SACK_BASE_ALIGNED +
(tp->eff_sacks * TCPOLEN_SACK_PERBLOCK));
        }
```

Now that we know the size of the TCP header, we adjust the skb to allow for sufficient space.

```
        th = (struct tcphdr *) skb_push(skb, tcp_header_size);
        skb->h.th = th;
        skb_set_owner_w(skb, sk);
```

At this point, the TCP header is built, space for the checksum field is reserved, and the header size is calculated. Some of the header fields used to build the header are in inet_opt structure, some are in the TCP control buffer, and some are in the TCP options structure.

```
        th->source      = inet->sport;
        th->dest        = inet->dport;
        th->seq         = htonl(tcb->seq);
        th->ack_seq     = htonl(tp->rcv_nxt);
        *(((__u16 *)th) + 6) = htons(((tcp_header_size >> 2) << 12) |
                                     tcb->flags);
```

The advertised window size is determined. If this packet is a **SYN** packet; otherwise, the window size is scaled by calling tcp_select_window.

```
if (tcb->flags & TCPCB_FLAG_SYN) {
    th->window = htons(tp->rcv_wnd);
}  else {
    th->window = htons(tcp_select_window(sk));
}
```

If urgent mode is set, we calculate the urgent pointer and set the URG flag in the TCP header.

```
if (tp->urg_mode && between(tp->snd_up, tcb->seq+1,
     tcb->seq+0xFFFF)) {
    th->urg_ptr = htons(tp->snd_up-tcb->seq);
    th->urg = 1;
}
```

Here we actually build the TCP options part of the packet header. If this is a **SYN** segment, we include the window scale option. If not, it is left out. We check to see if we have a timestamp, window scaling or SACK option from the sysctl_flags set earlier. We call tcp_syn_build_options to build the options with window scaling, and we call tcp_build_and_update_options is for non-SYN packets and do not include window scaling.

```
if (tcb->flags & TCPCB_FLAG_SYN) {
    tcp_syn_build_options((__u32 *)(th + 1),
                          tcp_advertise_mss(sk),
                          (sysctl_flags & SYSCTL_FLAG_TSTAMPS),
                          (sysctl_flags & SYSCTL_FLAG_SACK),
                          (sysctl_flags & SYSCTL_FLAG_WSCALE),
                          tp->rcv_wscale,
                          tcb->when,
                                tp->ts_recent);
}  else {
    tcp_build_and_update_options((__u32 *)(th + 1),
                                  tp, tcb->when);
```

We call the TCP_ECN_send to send explicit congestion notification. This TCP modification [RFC3168] changes the TCP header to make it slightly incompatible with the header specified by RFC 793.

```
    TCP_ECN_send(sk, tp, skb, tcp_header_size);
}
```

We calculate the checksum. We check the flags to see if we are sending an ACK or sending data and we update the delayed acknowledgment status depending on whether we are sending data or an ACK. See Section 8.9.4 for information about the delayed acknowledgment timer. If we are sending data, we update the congestion window and mark the send timestamp. In addition, we increment the counter to indicate the number of TCP segments that have been sent.

```
tp->af_specific->send_check(sk, th, skb->len, skb);
if (tcb->flags & TCPCB_FLAG_ACK)
    tcp_event_ack_sent(sk);
```

```
        if (skb->len != tcp_header_size)
            tcp_event_data_sent(tp, skb);
        TCP_INC_STATS(TcpOutSegs);
```

Now we call the actual transmission function to send this segment. The segment will be queued up for processing by the next stage whether the packet has an internal or an external destination.

```
        err = tp->af_specific->queue_xmit(skb);
        if (err <= 0)
            return err;
        tcp_enter_cwr(tp);
```

A return of a value less than zero tells the caller that the packet is dropped. We return all errors except explicit congestion notification, which doesn't indicate to the caller that the packet was dropped.

```
        return err == NET_XMIT_CN ? 0 : err;
    }
#undef SYSCTL_FLAG_TSTAMPS
#undef SYSCTL_FLAG_WSCALE
#undef SYSCTL_FLAG_SACK
    return -ENOBUFS;
}
```

8.8 Some Key TCP Data Structures

This section describes some important structures used in TCP. Most of the data structures are allocated on a per-socket or per-connection basis. The longest and most complicated of these structures is the TCP Options structure discussed in Section 8.8.2. This structure holds the TCP options and most of the variables used to maintain the TCP state machine and is accessed through the sock structure.

8.8.1 TCP Control Buffer

TCP is completely asynchronous. All socket-level writes are insulated from the actual packet transmission. It allocates socket buffers to hold the data as the application writes data into the socket. Each packet requires control information, which is passed from the tcp_sendmsg function to the transmission part of TCP. The socket buffer is described in Chapter 7. The TCP control block structure, tcp_skb_cb is defined in the file *linux/include/net/tcp.h*.

```
struct tcp_skb_cb {
    union {
```

Inet_skb_parm and inet6_skb_parm hold the IP options from an incoming packet, inet_skb_parm is for IPv4, and inet6_skb_parm are the options for IPv6.

```
        struct inet_skb_parm        h4;
#if defined(CONFIG_IPV6) || defined (CONFIG_IPV6_MODULE)
        struct inet6_skb_parm       h6;
#endif
```

```
} header;
```

Seq is the sequence number for the outgoing packet. End_seq is the ending sequence number so far, the sequence number, plus one for the **SYN** packet, plus one for the **FIN,** plus the length of this segment.

end_seq = seq + **SYN** + **FIN** + length of the current segment.

When is used for calculating RTT.

```
__u32          seq;
__u32          end_seq;
__u32          when;
```

Flags contains the TCP flags field in the TCP header. The values in this field should match bit for bit the flags in the actual TCP header; Table 8.3 lists the values. The last two items in the table are for explicit congestion control and are not defined in the original TCP specification [RFC 793]. They actually are taken from the 6-bit "reserved" area of the TCP header.

```
__u8           flags;
```

The sacked field holds the state flags for the Selective Acknowledge (SACK) and Forward Acknowledge (FACK) states. The possible states and their values are listed in Table 8.4.

```
__u8           sacked;
```

Table 8.4: Sacked State Flags

Flag	**Value**	**Purpose**
TCPCB_SACKED_ACKED	1	The data in the segment pointed to by this skb has been acknowledged by a SACK block.
TCPCB_SACKED_RETRANS	2	The segment needs to be retransmitted.
TCPCB_LOST	4	The segment is “lost.”
TCPCB_TAGBITS	TCPCB_SACKED_ACKED | TCPCB_SACKED_RETRANS| TCPCB_LOST	Combination of the previous three flags indicating that the segment is a tagged segment.
TCPCB_EVER_RETRANS	0x80	Indicates whether this segment was ever re-transmitted.
TCPCB_RETRANS	TCPCB_SACKED_RETRANS | TCPCB_EVER_RETRANS	Indicates that this is a retransmitted segment for any reason.

Table 8.4: Sacked State Flags

Flag	Value	Purpose
TCPCB_URG	0x20	Indicates that the TCP urgent pointer is advanced.
TCPCB_AT_TAIL	TCPCB_URG	Same as TCPCB_URG.

The next field, urg_ptr, is the value of the TCP header urgent pointer. The TCPCB_FLAG_URG must be set when this field is valid. The last field, ack_seq, is equivalent to the acknowledgment field in the TCP header.

```
    __u16         urg_ptr;
    __u32         ack_seq;
} ;
```

8.8.2 TCP Options Structure

The sock structure, discussed in Chapter 5, contains the support necessary to maintain connection state. Linux TCP, however, implements the TCP options structure, tcp_opt, which is defined in the file *linux/include/linux/tcp.h*. In addition to TCP options, it contains the send and receive sequence variables, TCP window management, and everything to manage slow start and congestion avoidance. Tcp_opt is presented in its own section because of its complexity and because it contains fields to help manage many aspects of the TCP protocol.

Tcp_opt is allocated as part of the sock structure for a SOCK_STREAM type socket. See Chapter 5 for more details about how sockets are allocated.

```
struct tcp_opt {
    int     tcp_header_len;
```

Pred_flags is used to determine if TCP header prediction should be done. Header prediction is used for fast path TCP reception. See Chapter 10, Section 10.7, for more details about how this field is used.

```
    __u32     pred_flags;
```

The following are the send and receive sequence variables [RFC 793]. These variables govern the sequence numbers exchanged between peers during data exchange through a connection. Rcv_next is the "receive next" sequence variable, **RCV.NXT** [RFC793]. It represents the next sequence number that is expected in incoming segments.

```
     __u32   rcv_nxt;
```

Send next sequence variable is **SND.NXT**. It is the next sequence number to be sent.

```
     __u32   snd_nxt;
```

This is the send unacknowledged sequence variable, **SND.UNA**, the oldest unacknowledged sequence number. It should be the next byte for which we should receive an **ACK**.

```
__u32   snd_una;
```

Snd_sml is the last byte in the most recently transmitted small packet.

```
__u32   snd_sml;
```

Rcv_tstamp is the timestamp of the last received acknowledge. It is used for maintaining keepalives when the SO_KEEPALIVE socket option is set and for calculating RTT.

```
__u32   rcv_tstamp;
```

Lsndtime is the timestamp when transmitted data was last sent. This is used for restart.

```
__u32   lsndtime;
```

The ack structure is for controlling delayed acknowledgment. Delayed acknowledgment is implemented to increase TCP efficiency by holding back on acknowl edg ing received data until there is return data to carry the **ACK**. This reduces the number of **ACK** s transmitted by combining the **ACK** with an outgoing data packet wherever possible. A comprehensive explanation of delayed acknowledgments is in Stevens, Section 19.3 [STEV94]. Pending contains the quick acknowledge state, shown in Table 8.5. Quick is the scheduled number of quick acknowledgments. If the field pingpong contains the value one, the normal delayed acknowledgment mode is enabled. When pingpong is zero, quick acknowledgment mode is enabled and **ACK** s are set as soon as possible. Blocked indicates that sending of an **ACK** was blocked for some reason.

```
struct {
    __u8      pending;
    __u8      quick;
    __u8      pingpong;
    __u8      blocked;
```

Table 8.5: TCP Acknowledge State Values, tcp_ack_state_t

State	Value
TCP_ACK_SCHED	1
TCP_ACK_TIMER	2
TCP_ACK_PUSHED	4

Ato is the calculated delayed acknowledge timeout interval, and timeout holds the actual timeout value for the delayed acknowledgment. Lrcvtime holds the time the last data packet was received.

```
    __u32      ato;
    unsigned long timeout;
    __u32      lrcvtime;
```

Last_seg_size is the size of the most recent incoming segment. Rcv_mss holds the best guess of the received MSS of the peer machine.

```
        __u16     last_seg_size;
        __u16     rcv_mss;
}  ack;
```

The ucopy structure holds the prequeue for fast copying of data from packets to application space. For more information, see Chapter 10, "Receiving the Data in the Transport Layer," Section 10.5.2, TCP Prequeue processing.

```
    struct {
        struct sk_buff_head   prequeue;
        struct task_struct    *task;
        struct iovec          *iov;
        int                   memory;
        int                   len;
    }  ucopy;
```

Snd_wl1 holds a received window update sequence number, snd_wnd is the maximum sized window that the peer is willing to receive, **SND.WND** variable [RFC 793]. In addition, max_window is the largest window received from the peer during the life of this connection.

```
    __u32   snd_wl1;
    __u32   snd_wnd;
    __u32   max_window;
```

Pmtu_cookie is set to the last PMTU (Path Maximum Transmission Unit) for the connection referenced by this socket, mss_cache is the current sending MSS, and mss_clamp is the maximum MSS negotiated at connection setup.

```
    __u32   pmtu_cookie;
    __u16   mss_cache;
    __u16   mss_cache_std;
    __u16   mss_clamp;
```

Ext_header_len is the TCP/IP packet header length, including IP and IP options. Ca_state holds the fast retransmit machine states listed in Table 8.6.

```
    __u16   ext_header_len;
    __u16   ext2_header_len;
    __u8    ca_state;
```

Table 8.6: Tcp_ca_state, Fast Retransmit States

State	Value
TCP_CA_Open	0
TCP_CA_Disorder	1
TCP_CA_CSR	2
TCP_CA_Recovery	
TCP_CA_Loss	4

Retransmits contains the number of unrecovered RTOs (Retransmit Timeouts). Reordering is a packet reordering metric representing the maximum distance that a packet can be displaced in the stream. Queue_shrunk indicates that the write queue has been reduced in size; a packet has been removed from the wmem_queued socket buffer queue.

```
    __u8    retransmits;
    __u8    reordering;
    __u8    queue_shrunk;
```

The following field is for the TCP_DEFER_ACCEPT socket option, which when set allows an application program to sleep until data arrives on a socket. The application program is awakened when data arrives. This option is expressed in seconds, but is converted into the number of retries and set in the defer_accept field.

```
    __u8    defer_accept;
```

TCP must measure round-trip time (RTT) to support timeout and retransmission based on the Van Jacobson algorithm [JACOB88]; also refer to the Karn & Partridge algorithm [KARN91]. Stevens has a discussion of RTT measurement in Sections 21.3 and 21.4 [STEV94]. The next few fields in the TCP options structure are used for measuring RTT. Backoff is the amount of time to back off before retransmitting, srtt is the smoothed RTT value, and mdev is the medium deviation. Mdev_max is the maximum medium deviation for the most recent RTT period, and rttvar is the smoothed value of mdev_max. Rtt_seq is the sequence number to update rttvar, and rto is the retransmit timeout.

```
    __u8    backoff;
    __u32   srtt;
    __u32   mdev;
    __u32   mdev_max;
    __u32   rttvar;
    __u32   rtt_seq;
    __u32   rto;
```

The next three fields are for calculating the number of packets that have been transmitted but not acknowledged—known as *packets in flight*. Packets_out is the amount of unacknowledged data that has been sent calculated from the number of bytes of data divided by the segment size. Left_out is the number of packets that have arrived out of order, plus the number of packets that have been lost. Retran_out is the number of retransmitted segments.

```
    __u32   packets_out;
    __u32   left_out;
_    ___u32   retrans_out;
```

The following six fields are used for slow start and congestion control as specified by the Nagle algorithm, [RFC 896], the Karn and Partridge algorithm [KARN91], and the Von Jacobson algorithm [JACOB88]. In addition, Section 3.2 RFC 2861, contains suggested pseudo-code for reducing the congestion window. Snd_ssthresh is the slow start size threshold. Snd_cwnd is the sending side congestion window, and snd_cwnd_cnt is the linear increase counter. Snd_cwnd_clamp is the maximum allowable value for snd_cwnd. Snd_cwnd_used is the window used variable, and snd_cwnd_stamp is the send congestion window timestamp.

```
__u32   snd_ssthresh;
__u32   snd_cwnd;
__u16   snd_cwnd_cnt;
__u16   snd_cwnd_clamp;
__u32   snd_cwnd_used;
__u32   snd_cwnd_stamp;
```

These three fields are for timers used widely on both send and receive sides. Timeout is a variable to hold the retransmission timer value, retransmit_timer is the actual timer for TCP retransmit, and delack_timer is the delayed acknowledge timer. Refer to Section 8.6 for more information about the TCP transmit timers.

```
unsigned long       timeout;
 struct timer_list   retransmit_timer;
 struct timer_list   delack_timer;
```

The socket buffer queue, out_of_order_queue, holds received out of order segments. Af_specific is a pointer to the AF_INET family-specific operations for TCP. Send_head points to the socket buffers queued for transmitting.

```
struct sk_buff_head out_of_order_queue;
struct tcp_func     *af_specific;
struct sk_buff      *send_head;
```

Rcv_wnd is the current receive window size, which is the **RCV.WND** variable in RFC 793. Rcv_wup, receive window update, is the current advertised window size. Write_seq is the send side sequence number, actually equal to the end of the data in the last send buffer, plus one byte. Pushed_seq is the last pushed sequence number. Copied_seq is the position of the start of received data that has not yet been read by the user process.

```
 __u32   rcv_wnd;
__u32   rcv_wup;
__u32   write_seq;
__u32   pushed_seq;
__u32   copied_seq;
```

The following fields are used for received TCP options. The TCP options are listed in Table 8.7. Some options are received only in SYN packets, and others can be found in any data packet. The next field, tstamp_ok, indicates that a timestamp was received in the SYN packet. Wscale_ok indicates that the window scale option was received in a SYN packet. Window scaling increases TCP efficiency by decreasing the number of ACKs. The window scale option specifies a shift count, which is the number of bits left to shift the window size. This allows the 16-bit window field in the TCP header to represent a much larger window than 65535. See Stevens, Section 24.4 for a discussion of window scaling [STEV94]. Sack_ok indicates that the TCPOPT_SACK option was received in a SYN packet, and saw_tstamp indicates that a timestamp option, TCPOPT_TIMESTAMP, in the most recent packet was seen and processed.

```
char    tstamp_ok,
        wscale_ok,
        sack_ok;
char    saw_tstamp;
```

Table 8.7: TCP Options

Option Option	Kind	Length	Description
TCPOPT_EOL	0	0	End of options.
TCPOPT_NOP	1	0	Padding
TCP_OPT_MSS	2	4	Segment size negotiation.
TCPOPT_WINDOW	3	3	Window scaling.
TCPOPT_SACK_PERM	4	2	Selective acknowledgments are permitted. See RFC 2018.
TCPOPT_SACK	5	variable	Actual selective acknowledg ments are included in the options.
TCPOPT_TIMESTAMP	8	10	Sender can include this option in any segment. The sender's timestamp value is echoed back in an ACK packet so sender can calculate RTT.

Snd_wscale holds the window scaling factor, which was received from the peer. Rcv_wscale is the window scaling factor that is sent to the peer. Both these fields are used with the window scaling option, TCPOPT_WINDOW. Nonagle is set to the number one when it holds the TCP_NODELAY socket option, or set to the number two when it holds the TCP_CORK option. These two socket options are mutually exclusive, and both have the effect of disabling the Nagle algorithm but in opposite ways. When the TCP_NODELAY option is set, data is sent out as soon as possible in small segments without waiting for enough data to fill a full-sized segment and without waiting for any outstanding small segments to be acknowledged. However, if the TCP_CORK option is set, TCP holds on to the small segments as long as the option remains set. The variable is named nonagle because it disables the Nagle algorithm, which allows only one small segment to be outstanding. No other segments can be sent until the first segment is acknowledged by the peer. For a complete discussion of the Nagle algorithm, refer to RFC 896 or review Section 19.4 in Stevens where the algorithm is discussed in detail [STEV94].

```
__u8    snd_wscale;
__u8    8rcv_wscale;
__u8    nonagle;
```

Keepalive_probes holds the value of the TCP_KEEPCNT socket option. This option, unique to Linux, specifies the number of keepalive probes that are sent before the connection is dropped. This option is applicable when the SO_KEEPALIVE socket option is also set.

```
__u8    keepalive_probes;
```

The next four fields are to support the Protection Against Wrapped Sequence Numbers (PAWS) algorithm [RFC 1323] and Round Trip Time Measurement (RTTM). For a complete analysis of PAWS, see RFC 1323. In addition, Stevens has a complete discussion of RTTM in Section 21.4, and a discussion of the timestamp option and PAWS in Sections 24.5 and 24.6 [STEV94]. Rcv_tsval is the received time stamp value and rcv_tsecr is the value of the received time stamp echo. Ts_recent is the specific received time stamp that will be echoed back in the next

timestamp option to be sent. Ts_recent_stamp is used for aging; it is the time that the received timestamp, ts_recent, was stored.

```
__u32   rcv_tsval;
__u32   rcv_tsecr;
__u32   ts_recent;
long    ts_recent_stamp;
```

The following five fields are for Selective Acknowledgement (SACKS) [RFC 2018]. User_mss is the MSS requested by the application program, and dsack in dicates that a Duplicate SACK (D-SACK) is scheduled to be sent; see RFC 2018, Section 4. Eff_sacks is the size of the array of SACK blocks to send in the next packet. Duplicate_sack is the D-SACK block, and the selective_acks array contains the actual SACKS.

```
__u16   user_mss;
__u8    dsack;
__u8    eff_sacks;
struct tcp_sack_block     duplicate_sack[1];
struct tcp_sack_block     selective_acks[4];
```

Window_clamp is the maximum-sized TCP window to advertise, and rcv_ssthresh is the current window clamp. Rcv_ssthresh is the maximum window size used during slow start phase. Probes_out is the number of zero window probes, which are unanswered zero window probes that have been sent. Stevens [STEV94] Chapter 22 discusses window probes as part of the TCP persist timer. Num_sacks is the number of SACK blocks. Advmss is the advertised MSS. Syn_retires holds the value of the TCP_SYNCNT option, the number of allowed SYN retries.

```
__u32   window_clamp;
__u32   rcv_ssthresh;
__u8    probes_out;
__u8    num_sacks;
__u16   advmss;
__u8    8syn_retries;
```

Ecn_flags are for Explicit Congestion Notification (ECN). This field contains the ECN status bits, which correspond to the last two bits in byte 13 of the TCP header. RFC 3168 contains the specification for ECN. See Section 6 in RFC 3168 for a discussion of the status flags.

```
__u8    ecn_flags;
```

Prior_ssthreash holds the slow start threshold value saved from the start of the congestion recovery phase. It is the previous value of the ssthresh. Lost_out is the number of lost packets, the number of segments that were sent but not acknowledged. Sacked_out is the number of segments that were sent by this side and have arrived at the receiver but have been acknowledged with SACKs. Fackets_out is the number of transmitted packets that have been Forward Acknowledged (FACKed). High_seq is set to the value of snd_nxt when congestion is detected or the loss state is entered. For more information on FACKs, see Mathis and Mahdavi's papers on forward acknowledgement [MATH96] and [MATH97].

```
__u16   prior_ssthresh;
__u32   lost_out;
```

```
__u32   sacked_out;
__u32   fackets_out;
__u32   high_seq;
```

Retrans_stamp is set to the timestamp of the most recent retransmit. It is used in the **SYN_SENT** state to retrieve the time the last **SYN** packet was sent. Undo_marker indicates when tracking of retransmits have started; it is set to the snd_una variable. Undo_retrans is the number of retransmits that need to be undone if possible. Urg_seq is the sequence number of a received urgent pointer, and urg_data contains the number of the saved byte of Out-of-Band (OoB) data, plus the associated control flags.

```
__u32   retrans_stamp;
__u32   undo_marker;
int     undo_retrans;
__u32   urg_seq;
__u16   urg_data;
```

It indicates if the MSG_OOB flag is set. Snd_up is the urgent pointer to be sent.

```
__u32   snd_up;
```

The following four fields are used for listening for connections. The first two are for the SYN acknowledge hash table. The SYN table is so that sockets in listening mode can efficiently match incoming connection requests. Listen_opt contains the SYN table, syn_table, a hash table of open_requests. Syn_wait_lock is a mutual exclusion (mutex) lock of the SYN table. There is already a master lock at the socket level, but this additional lock is acquired in read mode from tcp_get_info and acquired in write mode wherever the syn_table is updated. The next two fields, accept_queue and accept_queue_tail, hold the list of established connections of sockets that are children of this open socket.

```
rwlock_tsyn_wait_lock;
struct tcp_listen_opt*listen_opt;
struct open_request*accept_queue;
struct open_request*accept_queue_tail;
```

Write_pending indicates that a socket-level write request is pending.

```
intwrite_pending;
```

Keepalive_time is the amount of time the connection is allowed to remain idle before keepalive probes are set. It is initialized to two hours by the constant TCP_KEEPALIVE_TIME in file *tcp.h*. It can be changed with the TCP_KEEPIDLE socket option. Keepalive_intvl is the time between each transmission of a keepalive probe and is initialized to 75 seconds. It can be changed via the socket option, TCP_KEEPINTVL. Both of these TCP socket options are unique to Linux and therefore are not portable. Linger2 holds the value of another unique Linux TCP socket option, TCP_LINGER2. This value governs the lifetime of orphaned sockets in the **FIN_WAIT2** state. It overrides the default value of one minute in the TCP_FIN_TIMEOUT constant in file *tcp.h*.

```
unsigned int        keepalive_time;
```

```
    unsigned int        keepalive_intvl;
    int                 linger2;
```

Finally, the last field in the TCP options structure, last_synq_overflow, is for the support of syncookies, a security feature in Linux TCP. Syncookies consists of a mechanism to protect TCP from a particular type of denial-of-service attack (DoS) where an attacker sends out a flood of SYN packets. Syncookies are enabled via the tcp_syncookies sysctl value.

```
    unsigned long       last_synq_overflow;
} ;
```

8.9 TCP Timers

This chapter is about sending data, the transmit side of the TCP/IP stack. As discussed earlier, TCP maintains the connection state internally and requires timers to keep track of events. The TCP requires three timers to maintain the state on the transmit side in the protocol. These three timer functions could be implemented in one timer, but Linux uses three separate timers. In Chapter 6, we discussed the timer facility in Linux. When timers are initialized, they are given an associated function that is called when the timer goes off. Of course, each timer is completely re-entrant, and each timer function for TCP is passed a pointer to the sock structure. The timer uses the sock to know which connection it is dealing with The four timers are listed in Table 8.8.

Table 8.8: TCP Transmit Timers

Timer Name	**Timer Structure**	**Timer Function**	**Purpose**
TCP_TIME_RETRANS	tp->retransmit_timer	tcp_write_timer	Retransmit Timer
TCP_TIME_PROBE0	tp->retransmit_timer	tcp_write_timer	Zero window probe timer
TCP_TIME_DACK	tp->delack_timer	tcp_delack_timer	Delayed Acknowledg-ment Timer
TCP_TIME_KEEPOPEN	sk->timer	tcp_keepalive_timer	Keepalive Timer

The timer functions can be found in the file *linux/net/tcp_timer.c*. Both the data pointer and the timer function pointer for each of the retransmit timers are maintained in the TCP options part of the sock structure described in Section 8.4.2. The keepalive timer is maintained directly in the sock structure. The sock structure is shown in Chapter 5. In addition, functions to manage the timers, also in the file *linux/net/timer.c*, create, initialize, and delete all the timers. These functions are declared in file *linux/include/net/tcp.h*. The function tcp_init_xmit_timers initializes all the TCP timers.

```
void tcp_init_xmit_timers(struct sock *sk);
```

This function, tcp_clear_xmit_timers, clears all the TCP timers.

```
void tcp_clear_xmit_timers(struct sock *sk);
```

Linux TCP provides two functions to manage the individual retransmit, zero probe, and delayed acknowledgment timers. The first function, tcp_clear_xmit_timer, deletes the timer specified by the argument what, which specifies one of the timers from Table 8.8.

```
static inline void tcp_clear_xmit_timer(struct sock *sk, int what);
```

The second function, tcp_reset_xmit_timer, resets the timer to expire at the time specified by the argument when.

```
static inline void tcp_reset_xmit_timer(struct sock *sk, int what,
unsigned long when);
```

In addition, there are two separate functions to delete and reset the keepalive timers, tcp_delete_keep_alivetimer and tcp_reset_keepalive_timer.

```
extern void tcp_delete_keepalive_timer (struct sock *);
extern void tcp_reset_keepalive_timer (struct sock *, unsigned long);
```

Tcp_set_keepalive sets the keepalive timeout to the value, val.

```
void tcp_set_keepalive(struct sock *sk, int val)
```

8.9.1 TCP Write Timer

The TCP write timer serves two send-side timer purposes. The first purpose is the retransmission timer, which is set to the maximum time to wait for an acknowledgment after sending data. The other purpose is the window probe timer. Window probes are periodically sent from the send side of the connection once a zero window size is received from the peer. The probes are sent periodically to see if the window size has been increased, and the window probe timer is set to the maximum time to wait for a response to the window probe. The retransmission and zero window conditions do not occur simultaneously; therefore, both functions can be implemented with the same timer. The retransmission timer is set after sending any segment containing data, but the window probe timer is set after receiving an acknowledgment from the receiver with a window size of zero. Stevens devotes considerable time to the explanations of these two timers in two chapters. Chapter 21, “TCP Timeout and Retransmission” discusses the retransmission timer in detail, and Chapter 22, “TCP Persist Timer” discusses the window probe timer [STEV94].

When the timer expires, the function tcp_write_timer defined in file *linux/net/ipv4/tcp_timer.c* is executed with the argument, data, which points to the socket containing the newly expired timer.

```
static void tcp_write_timer(unsigned long data)
{
```

Sk is set to the sock structure pointed to by data, and tp is set to the TCP options structure in the sock structure.

```
    struct sock *sk = (struct sock*)data;
    struct tcp_opt *tp = tcp_sk(sk);
    int event;
```

For safety, the socket is locked. See Chapter 5 for details about the socket locking. If the socket is locked, we want to try again later so we set the timer value to 20 seconds and return.

```
    bh_lock_sock(sk);
    if (sock_owned_by_user(sk)) {
        if (!mod_timer(&tp->retransmit_timer, jiffies + (HZ/20)))
            sock_hold(sk);
        goto out_unlock;
    }
```

The socket state sk->state and the quick acknowledgment state tp->pending are checked to see if the socket is closed and there are no outstanding segments that have not yet been acknowledged.

```
    if (sk->state == TCP_CLOSE || !tp->pending)
        goto out;
```

If the timeout value is still in the future, the retransmit timer is reset to the value in the timeout field in the options structure. A zero return from mod_timer indicates that the timer was still pending. If the timer was still pending, the socket reference count is incremented and the function returns.

```
    if (time_after(tp->timeout, jiffies)) > 0) {
        if (!mod_timer(&tp->retransmit_timer, tp->timeout))
            sock_hold(sk);
        goto out;
    }
```

As discussed earlier, the write timer serves two purposes. It is used as the retransmit timer, TCP_TIME_RETRANS, or the window probe timer, TCP_TIME_PROBE0. We determine which role we are performing by looking at the pending field in the TCP options structure. If the current timeout is a retransmission timer expiring, we call the function tcp_retransmit_timer, but if the current timeout is a window prove timer expiring, we call tcp_probe_timer. These timer functions are discussed in the next two sections.

```
    event = tp->pending;
    tp->pending = 0;
    switch (event) {
        case TCP_TIME_RETRANS:
            tcp_retransmit_timer(sk);
            break;
        case TCP_TIME_PROBE0:
            tcp_probe_timer(sk);
            break;
    }
    TCP_CHECK_TIMER(sk);
out:
    tcp_mem_reclaim(sk);
out_unlock:
    bh_unlock_sock(sk);
```

```
    sock_put(sk);
}
```

8.9.2 TCP Retransmit Timer

Tcp_retransmit_timer function is called when the retransmit timer expires, indicating that an expected acknowledgment was not received. The function is invoked from the general TCP write timer function, tcp_write_timer, which was discussed earlier in Section 8.9.1.

```
static void tcp_retransmit_timer(struct sock *sk)
{
    struct tcp_opt *tp = tcp_sk(sk);
    if (tp->packets_out == 0)
        goto out;
    BUG_TRAP(!skb_queue_empty(&sk->sk_write_queue));
```

Next, we check to see if the connection should be timed out or if the sender has reduced the window size to zero with the socket still in the ESTABLISHED state. If a zero window has been received and the timestamp in the received packet indicates that the received packet is older than the maximum retransmit time, the next time the write timer is called it will become a zero window probe timer. The Congestion Avoidance (CA) processing state or loss state is entered by calling tcp_enter_loss.

```
    if (tp->snd_wnd == 0 && !sk->dead &&
        !((1<<sk->state)&(TCPF_SYN_SENT|TCPF_SYN_RECV))) {
#ifdef TCP_DEBUG
        if (net_ratelimit()) {
            struct inet_opt *inet = inet_sk(sk);
            printk(KERN_DEBUG "TCP: Treason uncloaked!
Peer %u.%u.%u.%u:%u/%u shrinks window %u:%u. Repaired.\ n",
                    NIPQUAD(inet->daddr), htons(inet->dport),
                     inet->num, tp->snd_una, tp->snd_nxt);
        }
#endif
```

If the received timestamp has aged more than two minutes, we indicate a write error, time out the socket, and drop the socket and the connection.

```
        if (tcp_time_stamp - tp->rcv_tstamp > TCP_RTO_MAX) {
            tcp_write_err(sk);
            goto out;
        }
```

The second parameter, now, of tcp_enter_loss is set to zero indicating that the loss state is being entered from a retransmit timeout. This part of congestion avoidance is discussed in more detail in Chapter 10. Next, we call tcp_retransmit_timer to try to retransmit the skb, which is at the head of the write queue by calling tcp_retransmit_skb. At this point, the connection is in a dubious state if the peer hasn't disappeared altogether, so __sk_dst_reset is called to reset the destination cache. Refer to Chapter 6, for information about the destination cache.

```
        tcp_enter_loss(sk, 0);
        tcp_retransmit_skb(sk, skb_peek(&sk->sk_write_queue));
```

```
        __sk_dst_reset(sk);
        goto out_reset_timer;
    }
```

Next, we call tcp_write_timeout to see if there has been a sufficient number of retries and to complete the processing for the last retry attempt.

```
    if (tcp_write_timeout(sk))
        goto out;
```

In the following section of code we increment the TCP statistics in the /proc filesystem. We check the congestion avoidance state, ca_state, field in the TCP option structure to see what sort of failure triggered the retransmit timeout. For more information about how to access TCP/IP statistics in /proc, refer to Chapter 6.

```
    if (tp->retransmits == 0) {
        if (tp->ca_state == TCP_CA_Disorder ||
             tp->ca_state == TCP_CA_Recovery) {
            if (tp->sack_ok) {
                if (tp->ca_state == TCP_CA_Recovery)
                    NET_INC_STATS_BH(TCPSackRecoveryFail);
                else
                    NET_INC_STATS_BH(TCPSackFailures);
            }  else {
                if (tp->ca_state == TCP_CA_Recovery)
                    NET_INC_STATS_BH(TCPRenoRecoveryFail);
                else
                    NET_INC_STATS_BH(TCPRenoFailures);
            }
        }  else if (tp->ca_state == TCP_CA_Loss) {
            NET_INC_STATS_BH(TCPLossFailures);
        }  else {
            NET_INC_STATS_BH(TCPTimeouts);
        }
    }
```

Now, the loss state is entered, congestion avoidance processing is initiated, and a retransmit is attempted.

```
    if (tcp_use_frto(sk)) {
        tcp_enter_frto(sk);
    }  else {
        tcp_enter_loss(sk, 0);
    }
    if (tcp_retransmit_skb(sk, skb_peek(&sk->write_queue)) > 0) {
```

If tcp_retransmit_skb returned a value greater than zero, it is because the low-level IP transmit function failed due to a local transmission problem such as a busy device driver. The retransmit timer must be reset to try again later.

```
        if (!tp->retransmits)
            tp->retransmits=1;
        tcp_reset_xmit_timer(sk, TCP_TIME_RETRANS, min(tp->rto,
                             TCP_RESOURCE_PROBE_INTERVAL));
```

```
        goto out;
    }
```

The retransmit timeout, rto, in the TCP options structure is increased each time a retransmit occurs; it is actually doubled each time. The maximum value of the retransmit timeout is TCP_RTO_MAX, or 120 seconds, which is also the maximum value for RTT. The doubling of the retransmit timer is suggested by Van Jacobson in his paper on congestion avoidance [JACOB88]. The Round-Trip Time estimate (RTT) is not changed by the timeout. If the number of retransmits is greater than TCP_RETR1, which is 3, the destination cache is aged out.

```
    tp->backoff++;
    tp->retransmits++;
out_reset_timer:
    tp->rto = min(tp->rto << 1, TCP_RTO_MAX);
    tcp_reset_xmit_timer(sk, TCP_TIME_RETRANS, tp->rto);
    if (tp->retransmits > sysctl_tcp_retries1)
        __sk_dst_reset(sk);
out:;
}
```

8.9.3 Window Probe Timer

The window probe timer, tcp_probe_timer, also in file *linux/net/ipv4/tcp_timer.c*, is called from the generic TCP write-side timer function, tcp_write_timer. As shown in Section 8.9.1, when the pending event is a window probe timeout, tcp_probe_timer is called. The zero window timer is set when this side of the connection sends out a zero window probe in response to a zero window advertisement from the peer. We arrive at this function because the timer expired before a response was received to the zero window probe.

Tp is set to point to the TCP options structure for the sock structure sk. First, tcp_probe_timer sees if there is a valid zero window event, which can occur only if the last packet sent was a zero window probe. The packet_out and the send_head fields in the TCP options structure are checked for outstanding unacknowledged data or data segments in the process of being sent. If either of these conditions exists, we return because this can’t be a valid zero window probe event.

```
static void tcp_probe_timer(struct sock *sk)
{
    struct tcp_opt *tp = tcp_sk(sk);
    int max_probes;
    if (tp->packets_out || !tp->send_head) {
        tp->probes_out = 0;
        return;
    }
```

We set the maximum number of probes out, max_probes, to the value TCP_RETR2, which is defined as the number 15 in the file *linux/include/net/tcp.h*. We do a check to see if the socket is orphaned. Linux does not kill off the connection merely because a zero window size has been advertised for a long time, so we want to see if we need to keep the connection alive. Therefore, we check to see if the next retransmission timeout value, rto, will still be less than the maximum RTO, TCP_RTO_MAX. Max_probes is recalculated for the orphaned socket. A call to tcp_out_of_resources checks to make sure that the orphaned socket does not consume too much

memory, as it should not be kept alive forever. If the socket is to be killed off, tcp_probe_timer returns here.

```
    max_probes = sysctl_tcp_retries2;
    if (sk->dead) {
        int alive = ((tp->rto<<tp->backoff) < TCP_RTO_MAX);
        max_probes = tcp_orphan_retries(sk, alive);
        if (tcp_out_of_resources(sk, alive || tp->probes_out <= max_probes))
            return;
    }
```

Next, we check the number of outstanding zero window probes to see if it has exceeded the maximum number of probes in max_probes, which was a value cal culated previously. If so, an error condition is indicated and the function returns. Another window probe is sent only if all the previous checks have passed.

```
    if (tp->probes_out > max_probes) {
        tcp_write_err(sk);
    }  else {
        tcp_send_probe0(sk);
    }
}
```

8.9.4 Delayed Acknowledgment Timer

The purpose of delayed acknowledgment is to minimize the number of separate ACKs that are sent. The receiver does not send an ACK as soon as it can. Instead, it holds on to the ACK in the hopes of piggybacking the ACK on an outgoing data packet. The delayed acknowledgment timer is set to the amount of time to hold the ACK waiting for outgoing data to be ready. The function, tcp_delack_timer, defined in file, *linux/net/ipv4/tcp_timer.c*, is called when the delayed acknowledgment timer expires, indicating that we have now given up finding an outgoing packet to carry our ACK. The timer value is maintained between a minimum value, TCP_DELACK_MIN, defined as 1/25 second, and a maximum value, TCP_DELACK_MAX, defined as 1/5 second.

As in the other TCP timer functions, tcp_delack_timer is called with a pointer to the sock structure for the current open socket.

```
static void tcp_delack_timer(unsigned long data)
{
    struct sock *sk = (struct sock*)data;
    struct tcp_opt *tp = tcp_sk(sk);

    bh_lock_sock(sk);
```

If the socket is locked, the timer is set ahead and an attempt is made later.

```
    if (sock_owned_by_user(sk)) {
tp->ack.blocked = 1;
NET_INC_STATS_BH(DelayedACKLocked);
        if (!mod_timer(&tp->delack_timer, jiffies + TCP_DELACK_MIN))
            sock_hold(sk);
```

```
        goto out_unlock;
    }
```

Tcp_mem_reclaim accounts for reclaiming memory from any TCP pages allocated in queues. If the socket is in the TCP_CLOSE state and there is no pending acknowledge event, we exit the timer without sending an ACK. Next, we check to see if we got here somehow even though the timer has not yet expired, in which case we exit.

```
tcp_mem_reclaim(sk);
    if (sk->state == TCP_CLOSE || !(tp->ack.pending&TCP_ACK_TIMER))
        goto out;
    if ((long)(tp->ack.timeout - jiffies) > 0) {
        if (!mod_timer(&tp->delack_timer, tp->ack.timeout))
            sock_hold(sk);

        goto out;
    }
    if (time_after(tp->ack.timeout, jiffies)) {
        if (!mod_timer(&tp->delack_timer, tp->ack.timeout))
            sock_hold(sk);
        goto out;
    }
```

Since the delayed ACK timer has fired, the pending timer event can be removed from the acknowledgment structure to indicate we are done processing the event.

```
    tp->ack.pending &= ~TCP_ACK_TIMER;
```

The field prequeue points to incoming packets that have not been processed yet. Since this timer went off before we could acknowledge these packets, they are put back on the backlog queue for later processing, and failure statistics are incremented for each of these packets.

```
    if (skb_queue_len(&tp->ucopy.prequeue)) {
        struct sk_buff *skb;
        NET_ADD_STATS_BH(TCPSchedulerFailed,
                         skb_queue_len(&tp->ucopy.prequeue));
        while ((skb = __skb_dequeue(&tp->ucopy.prequeue)) != NULL)
            sk->backlog_rcv(sk, skb);
        tp->ucopy.memory = 0;
    }
```

Here, we check to see if there is a scheduled ACK.

```
    if (tcp_ack_scheduled(tp)) {
```

If we have a scheduled ACK, it means that the timer expired before the delayed ACK could be sent out. We check the current acknowledgment mode in the pingpong field of the ack structure. If it is zero, we must be in quick acknowledgment mode, so we inflate the value of the acknowledgment timeout (ATO) in the ato field. This increases the amount of time until the next ACK timeout expires. However, if we are in delayed acknowledgment mode, we decrease the ATO to the minimum amount, TCP_ATO_MIN. In addition, we switch to fast acknowledgment mode by turning off delayed acknowledgment in the pingpong field. This will force the next

ACK to go out as soon as TCP_ATO_MIN time elapses without waiting for an outgoing data segment to carry the ACK.

```
        if (!tp->ack.pingpong) {
            tp->ack.ato = min(tp->ack.ato << 1, tp->rto);
        }  else {
            tp->ack.pingpong = 0;
            tp->ack.ato = TCP_ATO_MIN;
        }
```

Finally, we send the ACK and increase the delayed ACK counter. Next, we clean up and exit the timer.

```
        tcp_send_ack(sk);
        NET_INC_STATS_BH(DelayedACKs);
    }
    TCP_CHECK_TIMER(sk);
out:
    if (tcp_memory_pressure)
        tcp_mem_reclaim(sk);
out_unlock:
    bh_unlock_sock(sk);
    sock_put(sk);
}
```

8.9.5 Keepalive Timer

This timer function is actually used for two separate purposes. In addition to providing the keepalive timeout function, it is also used as a SYN acknowledge timer by a socket in a listen state. The keepalive timer function can serve these two separate purposes because keepalives are only sent for a connection that has already been established, but the SYN acknowledge timer is only active for connections in the **LISTEN** state.

TCP normally does not perform any keepalive function; keepalive polling is not part of the TCP specification. The keepalive is added outside the TCP specification for the use of some TCP application layer servers for protocols that don't do any connection polling themselves. For example, the telnet daemon sets the keepalive mode. Keepalive is enabled via the socket option SO_KEEPALIVE, and the timeout value associated with the timer is maintained in the keepopen field of the sock structure. It can also be set by using the system control value, sysctl_tcp_keepalive_time. The default value for the keepalive time is in the constant, TCP_KEEPALIVE_INTVL, defined in *linux/include/net/tcp.h* to 75 seconds.

The keepalive timer function, tcp_keepalive_timer in file *linux/net/ipv4/tcp_timer.c*, is called when the timeout value expires. As with the other TCP timer functions discussed in this section, a pointer to the sock structure is passed in via the parameter data. Tp is set to point to the tcp_opt structure in the sock.

```
static void tcp_keepalive_timer (unsigned long data)
{
    struct sock *sk = (struct sock *) data;
    struct tcp_opt *tp = tcp_sk(sk);
```

```
__u32 elapsed;
```

If the socket is currently in use, there is no need for keepalives, so we reset the timer value to 20 seconds and leave.

```
bh_lock_sock(sk);
if (sk->lock.users) {
    tcp_reset_keepalive_timer (sk, HZ/20);
    goto out;
}
```

Next, we must determine the state of the connection to see if the socket is in the **LISTEN**, **FIN_WAIT2**, or **TCP_CLOSE** connection state. This timer function is also used to maintain the SYNs and SYN_ACKs when a socket is in the **LISTEN** state. If the connection is in the **LISTEN** state, tcp_synack_timer is called to process SYN acknowledgments or the lack of them as the case may be.

```
if (sk->state == TCP_LISTEN) {
    tcp_synack_timer(sk);
    goto out;
}
```

If the socket is in the **FIN_WAIT2** state and the keepalive option is set, we want to do an abortive release of the connection when the timer expires instead of letting the connection terminate in an orderly fashion. A positive value in the linger2 field of the TCP options structure tells us that the TCP_LINGER2 socket option was set for this socket, so we want to maintain the connection, which is in **FIN_WAIT2** in an "undead" state before terminating the connection. If the linger2 field is less than zero, we shut down the connection immediately. Otherwise, the connection is aborted by calling tcp_send_active_reset, which sends the peer a **RST** segment.

```
if (sk->state == TCP_LISTEN) {
    tcp_synack_timer(sk);
    goto out;
}
if (sk->state == TCP_FIN_WAIT2 && sock_flag(sk, SOCK_DEAD)) {
    if (tp->linger2 >= 0) {
        int tmo = tcp_fin_time(tp) - TCP_TIMEWAIT_LEN;
        if (tmo > 0) {
            tcp_time_wait(sk, TCP_FIN_WAIT2, tmo);
            goto out;
        }
    }
    tcp_send_active_reset(sk, GFP_ATOMIC);
    goto death;
}
```

Next, we see if the connection state is in the **CLOSED** state and the keepalive mode is not set (keepopen field in the sock structure). If not, we get the value of the keepalive timer.

```
if (!sock_flag(sk, SOCK_KEEPOPEN)|| sk->state == TCP_CLOSE)
    goto out;
elapsed = keepalive_time_when(tp);
```

We check to see if the connection is actually alive and sending data. If it is, we just reset the keepalive timer without sending out the keepalive probe.

```
    if (tp->packets_out || tp->send_head)
        goto resched;
```

We are ready to send the keepalive as long as the elapsed time since the last probe is greater than the timeout value. The probes_out field in the TCP options structure is updated and the probe is sent by calling tcp_write_wakeup, which will wake up the socket and send the queued keepalive probe.

```
    elapsed = tcp_time_stamp - tp->rcv_tstamp;
    if (elapsed >= keepalive_time_when(tp)) {
        if ((!tp->keepalive_probes && tp->probes_out >=
             sysctl_tcp_keepalive_probes) ||
             (tp->keepalive_probes && tp->probes_out >=
              tp->keepalive_probes)) {
            tcp_send_active_reset(sk, GFP_ATOMIC);
            tcp_write_err(sk);
            goto out;
        }
```

If tcp_write_wakeup sent the probe out successfully, the probes_out counter is incremented and the configured keepalive timer value is reset to the configured value.

```
        if (tcp_write_wakeup(sk) <= 0) {
            tp->probes_out++;
            elapsed = keepalive_intvl_when(tp);
        }  else {
```

If the probe was not sent, the keepalive timer value is reset to 1BC2 second.

```
            elapsed = TCP_RESOURCE_PROBE_INTERVAL;
        }
    }  else {
        elapsed = keepalive_time_when(tp) - elapsed;
    }
    TCP_CHECK_TIMER(sk);
    tcp_mem_reclaim(sk);
```

This is where the timer is reset to get ready for sending the next keepalive probe.

```
resched:
    tcp_reset_keepalive_timer (sk, elapsed);
    goto out;
death:
    tcp_done(sk);
out:
    bh_unlock_sock(sk);
    sock_put(sk);
}
```

8.10 Summary

In this book, we have taken the approach of following the data as it is processed through the TCP/IP stack. In this chapter, we began following the data by looking at the input side of UDP and TCP. We covered each protocol's send function as it receives data from the socket layer for transmission. Of course, this is much simpler in UDP than in TCP. For TCP, we showed the state machine from the viewpoint of the receiver and looked at the basic TCP data structures and examined each of the TCP timers.

Chapter 9: The Network Layer, IP

According to the OSI layered model, it is the responsibility of the network layer to do packet addressing and routing. In the Linux implementation of the TCP/IP protocol suite, IP is responsible for maintaining the routing tables, called the Routing Policy Database (RPDB) in Linux. In addition, when a packet is received, IP must decide whether to route a packet out another interface or deliver it to one of the transport layer protocols, generally UDP or TCP. When a packet to be transmitted is received by IP from a transport layer protocol, it must determine if there is a route to the packet's destination and select the outgoing interface. Applying the MAC layer header is the responsibility of the network interface driver, but without knowing the details about how the packet is framed, IP provides the outgoing interface with a pointer to the packet's cached destination address. The IP layer implements fragmentation and re-assembly for packets that are too large for the physical limitations of the interface hardware. Another thing IP does is maintain the Time-To-Live (TTL) to make sure packets don't get endlessly routed in the Internet.

The Internet Control Message Protocol (ICMP) must be included in TCP/IP, as this is an integral part of the protocol suite. In addition to making routing decisions for unicast packets, IP must route multicast packets if multicast routing is configured in the Linux kernel. Therefore, both ICMP and IGMP are intrinsically linked with IP and can be thought of as part of the network layer.

9.1 Introduction

In this chapter, we introduce a little bit of routing theory, and then discuss the Linux IP routing mechanism in much more detail. In addition, we discuss other features of IP such as packet fragmentation. We also discuss ICMP, ARP, and IGMP and how these protocols interact with Linux IP. Generally, we explain things the same way we do elsewhere in the book, by following the path packets take as they are transmitted from the transport layer protocols, TCP and UDP through IP. In addition, we follow the packets as they are received by IP and passed up to the transport layer protocols. Along the way, we look at how the packet routing is done by looking at the Routing Policy Database (RPDB), which holds the routing policy information for Linux and explain how the RPDB and the route cache interact. We will show how IP input and output functions do RPDB lookups and how they access the routes in the cache to implement the routing decisions.

9.2 Routing Theory

At the simplest level, a machine running the IP protocol must determine the next hop for an output packet. It decides where to send an output packet based only on its destination address. If the sending machine knows that it can reach the destination through a LAN connection, this is a fairly simple matter of mapping the IP destination address to the hardware destination address. If the destination is not directly reachable, IP sends the packet to a gateway machine. This is called routing or packet forwarding and the information that is the basis of each packet forwarding decision is stored in the routing table. However, in the modern world of the Internet, things are quite a bit more complicated. Routing decisions can be made based on destination IP address

alone as in the simple case shown previously, or the decision can also be made on other packet fields such as the source IP address and the IP TOS field. Moreover, routing decisions can use other factors such as the output networking interface. In addition, the routing can be used with Network Address Translation (NAT) to actually change the destination addresses of the packet. Routing can be selected based on Quality of Service (QoS) where route lookups are done with the packet's ToS field in the IP header. The picture can be complicated with the addition of multicast routing and directed broadcast. Output packets may have multiple destinations, or they could be forwarded to a multicast address or a broadcast address out one or more output network interfaces. Another complication is that routes can be looked up based on traffic shaping values, and the lookup result used to select the particular queuing discipline for transmission at the queuing layer.

To support the real-world complexity of a configurable router, Linux provides more than a simple routing table and cache. It can be configured with multiple routing tables and a database of rules used to select the routing table for a particular route lookup. Setting up Linux routing configuration is done with application utilities that talk to the kernel through netlink sockets. We cover some of how netlink sockets work from the viewpoint of protocol module implementers. In this book, we concentrate on internal implementation rather than theory or network administration. However, there are a number of good references that provide theoretical background on routing or practical information on how to configure Linux-based routers using the application utilities. One of these is *Policy Routing Using Linux*, by Mathew G. Marsh [MARSH01].

9.3 IPV4 Routing, Routing Cache, and the Routing Policy Database

The Linux IP implementation maintains a routing table that is independent of the address format in IPv4 and IPv6 or other network layer protocols. This routing information is contained in the Routing Policy Database (RPDB). Two major components of the RPDB are the Forwarding Information Base (FIB) and the FIB rules. The FIB stores the routing information and the FIB rules are used to select a particular routing table when multiple tables are configured into the kernel. Linux has a route cache to help it make fast forwarding decisions using cached known routes. This helps the system performance when there is high packet volume and many simultaneous open sockets. The route cache itself is derived from the generic destination cache so it is able to make use of two other caches, the neighbor cache and the hardware header cache. The fundamental infrastructure for implementing the three caches is discussed in Chapter 6, "The Linux TCP/IP Stack." In this chapter, we will show how the routing cache is used in conjunction with the RPDB to make routing decisions. This chapter will discuss the initialization of the IP protocol, including ARP, ICMP, and IGMP initialization.

The RPDB supports both *static routing* and *dynamic routing*. From our internal point of view, there really isn't a difference because either type of routing information is added by external application programs. Static routing is where the RPDB is updated through application configuration programs such as iproute2 [HUBHOW02]. This program could be used either with host systems or gateway machines acting as routers. Dynamic routing adds external routing protocols such as OSPF [RFC 1583] and BGP [RFC 1771] that update the RPDB dynamically with information exchanged with other routers.

Among other things, the IP protocol is responsible for IP forwarding, also known as *packet routing*. For each incoming packet, IP must decide whether the packet is supposed to be consumed locally or sent out one of several network interfaces. In addition, for each outgoing packet, IP must decide through which network interface it should send the packet. Moreover, IP must provide enough information about the next destination of the outgoing packet so lower-layer protocols can construct the packet's link layer header. This is called the *next hop* or *gateway address*. The IP protocol and other Layer 3 protocols store the information used for this decision-making process in a dynamic data structure called a *routing table* or *routing cache*. The routing table implementation must provide the following capabilities:

- The routing table must accept and provide information from an external routing protocol.
- It must provide fast search capability on multiple fields.
- Both dynamic routes and static routes must be maintained.
- A timer is necessary to age out routes when no longer needed.
- A use count must be maintained for routes.
- Network and host routes must be differentiated.
- Direct and indirect routes must be differentiated.
- The routing table must support multiple network interfaces.
- Next-hop or gateway addresses must be kept in the routing table.

The routing table implementation in Linux has two components, a route cache for quick retrieval of temporary routes and a more "permanent" facility called the RPDB, which is used for lookup when a route can't be found in the cache. Another interesting feature of Linux is that the route cache is based on the generic destination cache. This is implemented in such a way that all packet forwarding decisions are symmetrical. Both output and input packets are routed based on destination cache entries. There are several advantages of this implementation. One is that routing of input packets to local addresses, loopback addresses, can be handled by lookups in the routing table. Once these local routes are in cache, fast internal forwarding decisions can be made. This way, internal messages such as netlink messages can be internally routed using the same mechanisms for routing packets sent to external destinations.

When the IP protocol needs to find the destination for either an outgoing or an incoming packet, it first checks the routing cache. If it can't find a route, it searches the FIB, which is where the routes in the RPDB are stored. The route cache is derived from the generic destination cache facility discussed in Chapter 6. The destination cache has routes in current use and routes that are stale but haven't aged out yet. In most practical host systems, these cached routes may be used to transmit and receive packets through open sockets. The FIB is the basic internal storage and lookup facility for the routing table. It used both for internal routing of packets intended for local consumption and routing of outgoing packets. The FIB also provides a way for applications outside the kernel to retrieve routes from the Linux kernel through routing sockets.

9.4 IP Protocol Initialization

IP protocol initialization begins with the AF_INET family initialization discussed in Chapter 6. This initialization is done by the function inet_init as part of its initialization of TCP, UDP, TCP slab cache, and other items. One of the steps performed by inet_init is to call ip_init to perform the initialization steps specific to the IP protocol.

```
void __init ip_init(void)
{
```

Dev_add_pack registers the IP protocol number, 0x0800, along with the packet handler function, so incoming IP packets are processed.

```
    dev_add_pack(&ip_packet_type);
```

Ip_rt_init initializes the routing tables and the FIB.

```
    ip_rt_init();
```

Inet_initpeers initializes the Internet peer structure, inet_peer, which hold long-lasting information about IPv4 destination addresses for other machines. Internet peers generate a unique number for the ID field of the IP packet for outgoing packets. Section 9.9 has more information about the ID field and how it is used with IP fragmentation. We call igmp_mc_proc_init to create an entry in the proc filesystem for information about IGMP and multicast routing if multicast is configured. Section 9.7 has more information about multicast routing.

```
    inet_initpeers();

#if defined(CONFIG_IP_MULTICAST) && defined(CONFIG_PROC_FS)
    igmp_mc_proc_init();
#endif}
```

9.5 The Route Cache

When a route is determined, the route entry is placed in the route cache for quick and easy access. This means that packets sent via the same route can be instantly routed once the route is known and placed in cache. We will show how the routing cache is searched for resolved routes before the FIB is checked.

A packet may have a destination within the local machine, its final destination may be at a locally reachable host, or it may be sent to a next-hop machine. Therefore, the route and destination caches are designed so the packet destination is transparent to the actual IP sending processes. The destination cache entry is interchangeable with a routing cache entry. Recall in Chapter 6, we discussed how the destination cache is used to find the link layer header by going through the dst field of the socket buffer. In addition to pointing to the destination cache, the same dst field can also point to the routing table entry. This provides IP with an efficient way to check the destination of an outgoing packet without having to look up the route for the packet or explicitly checking if the destination address has been resolved to a hardware address.

9.5.1 Route Cache Data Structures

The routing cache can be thought of as a front end to the FIB. However, it is optimized to provide fast path routing for open sockets with known destinations. The routing cache implementation is derived from the generic destination cache facility, discussed in Chapter 6. It

consists of a hash table, each containing routing table entries. The cache is designed for fast search with a simple key. Multiple hits at the same hash table location are known as *collisions*. The hash table implementation allows collisions because each hash table location can hold multiple routes.

The routing cache in IP is a single array of hash buckets, called the rt_hash_table, defined in file *linux/net/ipv4/route.c*. Each location in the array contains a pointer to a chain of routing table entries and a read-write lock, lock, so that each slot in the array can contain multiple routes. Each route that matches a table location is placed on a link list pointed to by chain.

```
struct rt_hash_bucket {
    struct rtable *chain;
    rwlock_t      lock;
}  __attribute__((__aligned__(8)));

static struct rt_hash_bucket  *rt_hash_table;
static unsigned               rt_hash_mask;
static int                    rt_hash_log;
```

The routing table, rtable, defined in file *linux/include/net/route.h*, is the fundamental data structure that holds each route in the route cache. Like many data structures in the Linux implementation of IPv4, it is essentially object oriented. This is how the routing cache can be considered to be derived from the generic destination cache facility. For example, the socket buffer structure, sk_buff, contains a pointer to the destination cache entry for an outgoing packet. This entry, dst, can contain a pointer to the routing cache entry for the packet. Therefore, the rtable structure is defined in such a way that the first few fields are identical to the fields of the destination cache, dst_entry.

```
struct rtable
{
```

We use a union so the pointer to the next rtable instance can be accessed as either a pointer to a destination cache entry through dst or a routing table entry pointer through rt_next.

```
    union
    {
        struct dst_entry      dst;
        struct rtable         *rt_next;
    }  u;
```

The rt_flags field contains the routing table flags, which are also used by the application interface to the routing table. These are defined in Table 9.2. Because there are multiple routes in a single hash bucket, the routes can clash. When garbage collection is done on the route cache, routes of lower value are cleaned out more aggressively when they clash with routes of higher value. The value of the routes is determined by the routing control flags.

```
    unsigned            rt_flags;
```

Rt_type is the type of this route. For example, it specifies whether the route is unicast, multicast, or a local route. The values for rt_type are in file *linux/include/linux/rtnetlink.h* and are shown in Table 9.1.

```
unsigned            rt_type;
```

Table 9.1: Route Types

Route Type	**Value**	**Explanation**
RTN_UNSPEC	0	Route is unspecified. This is a gateway or direct route.
RTN_UNICAST	0	This route is a gateway or direct route.
RTN_LOCAL	1	On input, accept packets locally.
RTN_BROADCAST	2	On input, accept packets locally as broadcast packets, and on output, send them as broadcast.
RTN_ANYCAST	3	Accept input packets locally as broadcast, but on output, send them as unicast packets.
RTN_MULTICAST	4	This is a multicast route.
RTN_BLACKHOLE	5	Drop these packets.
RTN_UNREACHABLE	6	The destination is unreachable.
RTN_PROHIBIT	7	This route is administratively prohibited.
RTN_THROW	8	The route is not in this table.
RTN_NAT	9	Translate this address using NAT.
RTN_XRESOLVE	10	Use external resolver to determine this route.

Rt_dst is the IP destination address, rt_src is the IP source address, and rt_iif is the route input interface index.

```
__u32               rt_dst;
__u32               rt_src;
int                 rt_iif;
```

The IP address of the gateway machine or neighbor is in the field rt_gateway.

```
__u32               rt_gateway;
```

The next field, fl, contains the actual hash key. The flowi structure is described later in the text.

```
struct flowi        fl;
```

The next few fields contain some miscellaneous cached information. The next field, rt_spec_dst, is the specific destination for the use of UDP socket users to set the source address. [RFC1122]

```
__u32               rt_spec_dst;
```

Internet peer structures are used for generating the 16-bit identification in the IP header. This is called the *long-living peer information*. Linux calculates the ID by incrementing a number for

each transmitted packet for a specific host. The Internet peer information for this route is accessed through the peer pointer.

```
    struct inet_peer    *peer;
```

The last two fields are used for NAT if it is configured into the kernel.

```
#ifdef CONFIG_IP_ROUTE_NAT
    __u32               rt_src_map;
    __u32               rt_dst_map;
#endif
} ;
```

Searches of the routing cache are done by locating a slot in the hash bucket array with a simple hash code. Next, a second search is done using a key to match a specific route by walking the list of routes accessed through the slot until rt_next is NULL. The second level of searching is done by looking for an exact match with the fl field in the rtable entry against information obtained from the incoming packet. fl contains the flowi structure or generic Internet flow. This structure is defined in file *linux/include/net/flow.h* and contains all the information to specifically identify a route.

```
struct flowi {
```

The next two fields identify the input and output interfaces. Iif is the input interface index; it is obtained from the ifindex field of the net_device structure for the network interface device from which a packet was received. Oif contains the index of the output interface. Generally, either iif or oif will be defined for a specific route and the other field will be zero.

```
    int     oif;
    int     iif;
```

This structure is generic, so we use a union to define parts for IPv4, IPv6, and DECnet. The IPv4 parts are explained later in the chapter.

```
    union {
        struct {
```

The next two fields, daddr and saddr, are the IP destination address, and the IP source address, respectively.

```
            __u32           daddr;
            __u32           saddr;
```

The next field, fwmark, is for firewall marks. This is part of traffic shaping and provides a way of aggregating different services on different ports so they can be directed through the same firewall hole.

```
            __u32           fwmark;
```

Tos is the IP header ToS field. The bits in the ToS field include the precedence bits and three bits for low delay, throughput, and reliability. IP doesn't define bit zero of ToS, but Linux uses it to identify a directly connected host. This bit is used so the same routing table searches functions for different purposes, including searches of the destination cache by the ARP protocol.

```
        __u8            tos;
```

Scope defines the scope or conceptual distance covered by this route. Values for scope are shown in Table 9.2.

```
        __u8            scope;
    }  ip4_u;
```

Table 9.2: Route Scope Definitions

Name	Value	Description
RT_SCOPE_UNIVERSE	0	This value indicates that the destination in the route can be anywhere.
RT_SCOPE_SITE	200	The destination address is within the site.
RT_SCOPE_LINK	253	The address is for a directly attached host, which is any machine that can be reached by a direct physical connection.
RT_SCOPE_HOST	254	This value is for addresses that are one of the local addresses on this host.
RT_SCOPE_NOWHERE	255	Destination address is nonexistent.

This is the end of the union for IPv4. Next comes IPv6. See Chapter 11, "Internet Protocol Version 6 (IPv6)," for more information about IPv6 routing.

```
    struct {
        struct in6_addr         daddr;
        struct in6_addr         saddr;
        __u32                   flowlabel;
    }  ip6_u;
```

The union for decnet, dn_u follows. It is beyond the scope of this book.

```
    struct {
        __u16           daddr;
        __u16           saddr;
        __u32           fwmark;
        __u8            scope;
    }  dn_u;
}  nl_u;
#define fld_dst         nl_u.dn_u.daddr
#define fld_src         nl_u.dn_u.saddr
#define fld_fwmark      nl_u.dn_u.fwmark
#define fld_scope       nl_u.dn_u.scope
#define fl6_dst         nl_u.ip6_u.daddr
#define fl6_src         nl_u.ip6_u.saddr
#define fl6_flowlabel   nl_u.ip6_u.flowlabel
```

These macros are for easy access to the IPv4 specific fields.

```
#define fl4_dst         nl_u.ip4_u.daddr
#define fl4_src         nl_u.ip4_u.saddr
#define fl4_fwmark      nl_u.ip4_u.fwmark
#define fl4_tos         nl_u.ip4_u.tos
#define fl4_scope       nl_u.ip4_u.scope

    __u8proto;
    __u8flags;
    union {
        struct {
            __u16 sport;
            __u16 dport;
        }  ports;

        struct {
            __u8  type;
            __u8  code;
        }  icmpt;

        struct {
            __u16 sport;
            __u16 dport;
            __u8  8objnum;
            __u8  objnamel; /* Not 16 bits since max val is 16 */
            __u8  objname[16]; /* Not zero terminated */
        }  dnports;

        __u32           spi;
    }  uli_u;
#define fl_ip_sport     uli_u.ports.sport
#define fl_ip_dport     uli_u.ports.dport
#define fl_icmp_type    uli_u.icmpt.type
#define fl_icmp_code    uli_u.icmpt.code
#define fl_ipsec_spi    uli_u.spi
} ;
```

The scope of the route is in the scope field of the flowi structure. We can think of the scope as the "distance" to the destination address. It is used for making decisions about how to route packets and how to classify the routes. The values for scope are shown in Table 9.2 and found in the file, *linux/include/linux/rtnetlink.h*. These are the same values used in the scope field of fib_result and the nh_scope field in the next-hop structure, fib_nhs. A user creating a special routing table application may define additional values for scope between zero and 199. Currently, though, the values most often used by Linux IPv4 are RT_SCOPE_UNIVERSE, RT_SCOPE_LINK, or RT_SCOPE_HOST. A higher number implies a "closer" destination except for RT_SCOPE_NOWHERE, which says the destination doesn't exist.

The bit definitions for the tos field in the flowi structure are shown in Table 9.3. These definitions are similar to the definitions for the Type of Service (ToS) field of the IP header. This is to support building a quick hash code based on ToS. However, one of the bits in the ToS field of the IP header has no counterpart in this structure. This bit is 0x02, IPTOS_MINCOST, which maps into bit 6 of the tos field of the IP header [RFC 795]. In addition, one bit that is not defined in the IP header is used by Linux for internal purposes, RTO_ONLINK, 0x01. All the IP ToS bit

definitions are found in the file *linux/include/linux/ip.h*, and RTO_ONLINK is defined in file *linux/include/net/route.h*.

Table 9.3: Bit Definitions for ToS Field of the Flowi Structure

Definition	Value	Purpose
IPTOS_LOWDELAY	0x10	This is the low delay bit defined by IP.
IPTOS_THROUGHPUT	0x08	Throughput bit defined for IP ToS.
IPTOS_RELIABILITY	0x04	Reliability bit in IP ToS field.
IPTOS_MINCOST	0x02	This bit is defined in the header file but does not appear to be referenced in the Linux source code.
RTO_ONLINK	0x01	This bit is used to identify a directly connected host reachable by link-level transmission.

9.5.2 Route Cache Utility Functions

As shown earlier, the first part of each route table entry, rtable, contains a destination cache entry structure, dst_entry. Dst_entry includes the dst_ops structure containing function pointers corresponding to operations for manipulating the cache entry. Each specific routing protocol that uses the destination cache facility can implement these functions. The generic destination cache is discussed in Chapter 6. The destination operation functions specific to IP are all implemented in *linux/net/ipv4/route.c* and are shown in Table 9.4.

Table 9.4: Dst_ops values for IP Route Cache

Field in dst_ops	IP Function or Value	Purpose
family	AF_INET	IPv4 address family.
protocol	ETH_P_IP	Protocol field in link header, 0x0800.
gc	rt_garbage_collect	Garbage collection function.
check	ipv4_dst_check	Releases destination cache entry.
destroy	ipv4_dst_destroy	Destructor function for routing table entries.
negative_advice	ipv4_negative_advice	This function is called if an entry becomes redirected or obsolete.
link_failure	ipv4_link_failure	Sends ICMP unreachable message and expires the route.
update_pmtu	ip_rt_update_pmtu	Update the MTU for the route.
entry_size	sizeof(struct rtable)	Specific size of the route table entry.

A couple of the functions in Table 9.4 deserve a closer look. The first of these is the garbage collection function, rt_garbage_collect, which is explained in Section 9.5.3. The next one is the destination check function, ipv4_dst_check, accessed through the check field of the dst_ops structure. All it does is call dst_release, the generic function for the destination cache.

```
static struct dst_entry *ipv4_dst_check(struct dst_entry *dst, u32 cookie);
```

The destroy function, ipv4_dst_destroy, is shown here.

```
static void ipv4_dst_destroy(struct dst_entry *dst);
```

It deletes the attached Internet peer, inet_peer, accessed through the rt field of the rtable structure.

The negative advice function, ipv4_negative_advice removes routes in the cache pointed to by dst.

```
static struct dst_entry *ipv4_negative_advice(struct dst_entry *dst);
```

First, it checks to see if the route is obsolete by checking the obsolete field of the dst_entry. If it is, it calls ip_rt_put to return the route to the slab cache. Otherwise, we want to delete all routes to this destination about which we received negative advice. We do this by checking to see if the entry has expired by checking the expires field in dst_entry, or if the route has been redirected, by the presence of the RTCF_REDIRECTED flag in the rt_flags field. If so, it recalculates the 32-bit hash and calls rt_del to delete all the routes in the matching hash bucket location.

In addition to the operation functions, a few routing cache functions are called directly. It is helpful to understanding the route cache to take a closer look at these functions, which are defined in route.h. Some of them are inlines and the other functions are defined in route.c. The first function, ip_rt_put, deletes a route by calling the generic dst_release function for the destination cache.

```
static inline void ip_rt_put(struct rtable * rt)
{
    if (rt)
        dst_release(&rt->u.dst);
}
```

The next function, rt_bind_peer, creates a peer entry for the route. This in effect adds information about the destination to the routing table.

```
void rt_bind_peer(struct rtable *rt, int create);
```

First, it creates a new inet_peer structure by calling inet_getpeer. Then, it sets the peer field in the rtable structure to the new peer and increments the reference count in the inet_peer structure while locking the route cache.

An application using SOCK_STREAM type sockets will need a connection with a remote peer before sending packets through an open socket. To establish the connection, the application calls connect on an open socket. One of the things done when the connection is established is to set up a route to the destination. In addition, applications using SOCK_DGRAM type sockets use the connect socket call to set up a route to the destination address before sending data. IP provides the function ip_route_connect for both TCP and UDP to establish a route. This function is defined in file *linux/include/net/route.h*.

```
static inline int ip_route_connect(struct rtable **rp, u32 dst, u32 src,
u32 tos, int oif, u8 protocol, u16 sport, u16 dport, struct sock *sk);
```

Actually, ip_route_connect function is only a front end for the main route resolving function, ip_route_output, which is discussed in Section 9.8.

A few other functions used with the routing tables deserve more than a mention. These functions aren't part of the interface to the route cache because they are static functions. However, they are used by route resolving functions such as ip_route_output. It will be helpful to look at these functions for understanding how route cache entries are created and used.

When a new route is ready to be used for transmitting packets, ip_route_output calls the function rt_intern_hash, defined in file *linux/net/ipv4/route.c*, to enter a route, rt, into the route cache.

```
static int rt_intern_hash(unsigned hash, struct rtable *rt,
struct rtable **rp);
```

Rt_intern_hash uses the hash parameter, as an index into the hash table, rt_hash_table. The index is for a location that most closely matches the new route. This location may contain zero, one, or more routes. Next, rt_intern_hash uses the flowi part in the rtable structure, rt, to look for an exact match with one of the routes at the hash table location. If it finds an exact match, it moves the matched route to the front of the list, increments its use count, and frees the new route rt. If there is not an exact match with any of the routes at the table location, the new route, rt, is put in the front of the list pointed to by chain at the hash table location. However, before the new route is put in the cache, we try to bind it to a neighbor cache location by calling arp_bind_neighbour. If it appears that the neighbor cache is full, we call rt_garbage_collect to remove routes until there is room for the new route, rt. If rt_intern_hash doesn't succeed in finding space, it returns the ENOBUFS error code.

Both of the two major route resolving functions, ip_route_output_slow and ip_route_input_slow, need to set up some specific information once they have a route to an external destination that does not point to a directly connected host. For example, IP allows the MTU for a route to be specified, and TCP has an option for negotiating the maximum segment size with the peer. The function called to set up this information is rt_set_nexthop.

```
static void rt_set_nexthop(struct rtable *rt, struct fib_result *res,
u32 itag);
```

9.5.3 Route Cache Garbage Collection

The route cache includes a Garbage Collection (GC) facility. The GC is based on the generic garbage collection in the generic destination cache. The route GC sets a specific function in the gc field of the destination cache operations structure. This is to override the function in the generic destination cache garbage collection. The route GC also has an expiration timer that acts as a fallback to completely delete routes from the cache after they age out. In addition to the GC timer, there is a timer for flush delay that governs the amount of time to wait after requesting that all the routes are flushed. The minimum and maximum values for route aging and some garbage collection values may be configured through sysctl. These configuration and control items will generally not be accessed by the typical user but are available for use by a more sophisticated routing application. In addition, the flush delay may be specified when a route flush is requested.

The garbage collection timer is called rt_periodic_timer and is defined in the file, *linux/net/ipv4/route.c*.

```
static struct timer_list rt_periodic_timer;
```

The function associated with this timer is rt_check_expire.

The flush timer is rt_flush_timer defined in the same file.

```
static struct timer_list rt_flush_timer;
```

The timer function for the flush timer is rt_run_flush. Both timers are initialized in the ip_rt_init function in the same file, which also initializes the route cache.

The default garbage collection timeout value, ip_rt_gc_timeout, is set to the configured value, NET_IPV4_ROUTE_GC_TIMEOUT or a default of five minutes. This value is set in the timer function, rt_check_expire, which executes once a minute but is varied slightly. The timer governs the maximum amount of time that aged-out routes are allowed to remain in the cache.

The timer function, rt_check_expire, checks for expired routes in the route cache, rt_hash_table.

```
static void rt_check_expire(unsigned long dummy);
```

In this function, we walk through the hash bucket locations in the route cache. If we find a nonempty location, we traverse the list of routes at that location. For each route found, we check the expires field of the destination cache entry to see if that route has aged out. We free each expired route by calling rt_free.

Rt_check_expire is a timer function and runs in the "bottom half" or interrupt context; therefore, it written to be efficient. The age of the route entry is calculated by subtracting the value in the lastuse field of dst_entry from the current time. We also look for broadcast and multicast routes. These routes have RTCF_BROADCAST and RTCF_MULTICAST set in the routing control flags, rt_flags. Broadcast and multicast routes are cleaned more aggressively, particularly if they clash with more specific routes. High value routes are those with the RTCF_REDIRECTED or RTCF_NOTIFY flags, and these are given preference over other routes when we have clashes in the route cache. The values for these flags and other routing control flags are shown in Table 9.7.

In addition to the route timer function, there is a garbage collection function, rt_garbage_collect, accessed through the gc field in the dst_ops field of the destination cache entry.

```
static int rt_garbage_collect(void);
```

Garbage collection is done whenever someone tries to allocate a route cache entry and there are too many routes in the table. Specifically, this happens when someone calls the destination cache allocation route, dst_alloc, and the gc_thresh value in the ops field of the destination cache is exceeded by the number of entries.

The goal of garbage collection is to keep the routing cache at an equilibrium point such that the number of aged-out entries is approximately the same as the number of new entries. Rt_garbage_collect maintains an internal static variable, expire, to control the number of expired entries that are allowed to remain in the cache. Garbage collection is expensive, so it should not be run unless really necessary. If the networking load is light, the route cache is allowed to keep a higher number of recent expired entries. As the number of routes comes closer to the size of the cache, fewer expired routes are allowed to remain. The elasticity value of the garbage collection can be changed through the routing sysctl value, NET_IPV4_ROUTE_GC_ELASTICITY and is initialized to a default value.

Each time Rt_garbage_collect is executed, it calculates a new goal from the number of entries in the cache and reduces it by an amount calculated by the elasticity value. This goal is not less than the number of entries in the cache, entries, minus the garbage collection threshold value, gc_thresh. Both of these parameters are from the dst_ops structure for the route cache. Once the goal is set, rt_garbage_collect goes through the cache and frees expired entries until the goal is reached. Because garbage collection is time consuming, the routine is kept from spinning by checking the current time, jiffies, to make sure that it hasn't executed for too long. It also checks to see if it is being called in a "bottom half" context, from a softirq. The number of expired entries freed depends on the goal, how long it has been since the entries have expired, and how full the route cache is.

Linux provides a separate timer for flushing the route cache. This is because the amount of time this operation takes could be quite a bit if there are a large number of routes in the cache. If the flush is done in the "bottom half" context, it starts a flush timer to delay the route cache flush. However, if the route cache flush is requested from "user mode," which is actually system call context, it is done immediately. This is a good thing particularly for embedded systems that need a predictable latency value. Real-time systems have latency requirements for other time critical kernel tasks running as soft_irqs that would be delayed if flushing the cache took too long.

Rt_cache_flush is the function that does the removal of cache entries. The parameter delay indicates the amount of time to wait before removing the cache entries.

```
void rt_cache_flush(int delay)
{
    unsigned long now = jiffies;
```

User_mode is zero if we are running in the "bottom half" context, and one otherwise.

```
    int user_mode = !in_softirq();
```

If the caller set delay to a negative number, we use the configured minimum delay time.

```
    if (delay < 0)
        delay = ip_rt_min_delay;
```

We must acquire the route cache lock because we could get called from different contexts.

```
    spin_lock_bh(&rt_flush_lock);
```

```
    if (del_timer(&rt_flush_timer) && delay > 0 && rt_deadline) {
        long tmo = (long)(rt_deadline - now);
```

If the flush timer is currently running, we reset the timer value to the current time plus the requested delay time. If user_mode is set, we set the delay to zero, which will run the timer function as soon as we exit.

```
        if (user_mode && tmo < ip_rt_max_delay-ip_rt_min_delay)
            tmo = 0;
        if (delay > tmo)
            delay = tmo;
    }
    if (delay <= 0) {
        spin_unlock_bh(&rt_flush_lock);
        rt_run_flush(0);
        return;
    }
    if (rt_deadline == 0)
        rt_deadline = now + ip_rt_max_delay;
    mod_timer(&rt_flush_timer, now+delay);
    spin_unlock_bh(&rt_flush_lock);
}
```

The next function, rt_run_flush, is the timer function that actually flushes the cache.

```
static void rt_run_flush(unsigned long dummy)
{
    int i;
    struct rtable *rth, *next;
    rt_deadline = 0;
```

Like the other route cache manipulation functions that update the contents of the route cache hash table, we must acquire the write lock. For each table location, we walk through the routes pointed to by chain and call rt_free to remove each route.

```
    for (i = rt_hash_mask; i >= 0; i—) {
        write_lock_bh(&rt_hash_table[i].lock);
        rth = rt_hash_table[i].chain;
        if (rth)
            rt_hash_table[i].chain = NULL;
        write_unlock_bh(&rt_hash_table[i].lock);
        for (; rth; rth = next) {
            next = rth->u.rt_next;
            rt_free(rth);
        }
    }
}
```

9.6 The RPDB, the FIB, and the FIB Rules

The Routing Policy Database (RPDB) consists of the Forwarding Information Base (FIB) and the FIB rules. The FIB is the core of Linux routing. Linux can be configured to support multiple

routing tables, but as a default it is configured with only two tables. In many common uses of the Linux kernel there is no need for policy-based routing. This would be the case with most embedded systems. Therefore, multiple tables need not be configured into the kernel. If multiple tables are not configured, there are two predefined global variables pointing to two tables, the local table and the main table. The local table contains routes to addresses that are locally assigned within the machine, such as addresses assigned to network interface devices and the loopback address. The main table contains routes to external machines. Both of these tables are declared in *linuxnet//ipv4/fib_frontend.c*.

```
struct fib_table *local_table;
struct fib_table *main_table;
```

If multiple tables are configured, there is an array of FIB table pointers numbered one through 256 (zero is an undefined FIB table index). Each table is referenced by an ID that serves as an index into the array. When multiple tables are defined, the main table and local table each are represented by two IDs that point to the last two locations in the array. RT_TABLE_MAIN is defined as 254, and RT_TABLE_LOCAL is 255. In addition, there is a default location if no ID is defined, RT_TABLE_DEFAULT, which is 253. The definition of the table locations are in *include/linux/rtnetlink.h* because when application programs such as routing daemons add and create routes, they are actually passing messages through netlink sockets to the FIB. As a new FIB table is created it will use an ID numbered between 1 and 252.

```
struct fib_table *fib_tables[RT_TABLE_MAX+1];
```

In addition to multiple FIB tables, Linux supports FIB rules that are used to select among the tables. Application programs will define FIB rules and how the rules will be used to do lookups in particular tables

9.6.1 FIB Internal Data Structures

Each location in the array, fib_tables, described later, contains a pointer to a FIB table structure, fib_table, defined in *linux/include/net/ip_fib.h*. Fib_table hides most of the complexity of the FIB. It consists largely of pointers to operation functions that are called by front-end functions. These internal operation functions are usually not called directly. The idea is to make the FIB as non-IP specific as possible. Someone implementing a network layer protocol could define his own FIB with its own operation functions that process the routing protocol's unique address format and routing rules. The FIB has what are called front-end functions for network layer protocols like IP to call as the interface to the FIB to create, delete, and otherwise manipulate routing entries. These front-end functions are explained later in this chapter. The application layer interface to the FIB is through rtnetlink sockets (routing sockets). This is how an application layer routing protocol updates the FIB by adding and deleting routes.

The fib_table structure mainly consists of the FIB operation function pointers. These function pointers are initialized when a specific fib_table is created. The initialization of the FIB is discussed in Section 9.5.2.

```
struct fib_table
{
```

The tb_id field is the table ID. If multiple tables are configured, it consists of one of the values between 1 and 255. If multiple tables are not defined, tb_id is set to either RT_TABLE_MAIN or RT_TABLE_LOCAL. Tb_stamp is not used.

```
    unsigned char tb_id;
    unsigned      tb_stamp;
```

Tb_lookup, tb_insert, and tb_delete are pointers to functions that look up, insert, and delete routes from the FIB, respectively.

```
    int           (*tb_lookup)(struct fib_table *tb, const struct flowi *flp,
                               struct fib_result *res);
    int           (*tb_insert)(struct fib_table *table, struct rtmsg *r,
                               struct kern_rta *rta, struct nlmsghdr *n,
                               struct netlink_skb_parms *req);
    int           (*tb_delete)(struct fib_table *table, struct rtmsg *r,
                               struct kern_rta *rta, struct nlmsghdr *n,
                               struct netlink_skb_parms *req);
```

The function tb_dump gets all the routes in the table. It called from inet_dump_fib and is used with rtnetlink.

```
    int           (*tb_dump)(struct fib_table *table, struct sk_buff *skb,
struct netlink_callback *cb);
```

The tb_flush operation removes all entries from the fib_table referenced by table.

```
    int            (*tb_flush)(struct fib_table *table);
```

This operation tb_select_default selects the default route.

```
    void           (*tb_select_default)(struct fib_table *table,
const struct rt_key *key, struct fib_result *res);
```

Tb_data is really an opaque pointer to the hash entries for this FIB table. It is manipulated by functions pointed by the other fields in the table. Tb_data is not accessed directly.

```
    unsigned char tb_data[0];
} ;
```

Another data structure used by the FIB is fib_info, also defined in file *linux/include/net/ip_fib.h*. It contains the data for routes to external unicast addresses that must go through a gateway. If a FIB entry contains a local or broadcast route, this data structure is not used.

```
struct fib_info
{
    struct fib_info     *fib_next;
    struct fib_info     *fib_prev;
    int                 fib_treeref;
    atomic_t            fib_clntref;
    int                 fib_dead;
    unsigned            fib_flags;
```

```
    int                 fib_protocol;
```

This field, fib_prefsrc, is the preferred source address. It is used as the "specific destination address" specified by RFC 1122 as part of UDP multihoming. It is used by UDP as the source address for response packets.

```
    u32                 fib_prefsrc;
    u32                 fib_priority;
    unsigned            fib_metrics[RTAX_MAX];
#define fib_mtu fib_metrics[RTAX_MTU-1]
#define fib_window fib_metrics[RTAX_WINDOW-1]
#define fib_rtt fib_metrics[RTAX_RTT-1]
#define fib_advmss fib_metrics[RTAX_ADVMSS-1]
```

Fib_nhs is the number of next hops. It should be one if this route has a gateway defined; otherwise, it will be zero. However, if multipath routing is configured, it can have a value greater than one. The next field, fib_power, is only used if multipath routing is configured.

```
    int                 fib_nhs;
#ifdef CONFIG_IP_ROUTE_MULTIPATH
    int                 fib_power;
#endif
```

Fib_nh contains information about the next hop or the list of next hops if there are more than one. Fib_dev is the network interface device that would be used to send a packet to the next hop.

```
    struct fib_nh       fib_nh[0];
#define fib_dev         fib_nh[0].nh_dev
} ;
```

The next data structure of interest is fib_nh, also in file *linux/include/net/ip_fib.h*. It is accessed by the routing mechanism to extract information about the next hop or gateway machine.

```
struct fib_nh
{
    struct net_device   *nh_dev;
    unsigned            nh_flags;
```

This field, nh_scope, is similar to the scope field in the flowi structure. It doesn't really define the scope of the route, but rather the conceptual distance to the destination machine; in this case, the gateway machine.

```
    unsigned char       nh_scope;
#ifdef CONFIG_IP_ROUTE_MULTIPATH
    int                 nh_weight;
    int                 nh_power;
#endif
```

The next field is the traffic class identifier, nh_tclassid, for the gateway. This is similar to the tclassid field in the destination cache entry, and is used primarily for traffic shaping.

```
#ifdef CONFIG_NET_CLS_ROUTE
```

```
    __u32               nh_tclassid;
#endif
```

Nh_oif is the index number of the output interface that would be used to reach the gateway machine.

```
    int                 nh_oif;
    u32                 nh_gw;
} ;
```

9.6.2 The FIB Rules

FIB rules are part of the RPDB and are most useful when the FIB is configured with multiple tables. When multiple tables are configured, in effect, we have multiple logical routers, and each router is represented by a FIB table instance. In a practical sense, the routing policy is a way of selecting one of the routing tables. When we are configured with multiple tables, we still have the two default tables, the local table containing local address information, and the main table containing the routing information. However, the other tables are accessed by using the FIB rules.

In this section, we show the fib_rule data structure and how FIB rules are used to select among the two "permanent" default routing tables. We also show how FIB rules are added and deleted and how a new routing table is created.

The fib_rule structure is defined in the source file *linux/net/ipv4/fib_rules.c*. It is not accessed directly from functions in other files.

```
struct fib_rule
{
```

FIB rules are implemented as a linked list through r_next. R_clntref is the use count, which must be atomically incremented.

```
    struct fib_rule     *r_next;
    atomic_t            r_clntref;
```

The next field, r_preference, contains the route preference value, and r_table is the index of the FIB table selected by this rule.

```
    u32                 r_preference;
    unsigned char r_table;
```

This field, r_action, has the same values as the route message type in the rtmsg structure. These values are defined in Table 9.1.

```
    unsigned char r_action;
    unsigned char r_dst_len;
    unsigned char r_src_len;
```

The next two fields are the source address and its netmask.

```
    u32           r_src;
    u32           r_srcmask;
```

These two fields are the destination address and the netmask for the destination.

```
    u32           r_dst;
    u32           r_dstmask;
```

The next field, r_srcmap, contains the address to which to map the source address. It is used where we have an address translation rule.

```
    u32           r_srcmap;
    u8            r_flags;
    u8            r_tos;
#ifdef CONFIG_IP_ROUTE_FWMARK
    u32           r_fwmark;
#endif
```

R_ifindex is index of the network interface device used in this rule.

```
    int           r_ifindex;
```

The next field is the traffic class identifier for this rule. It is used only if traffic class routing is configured in the kernel.

```
#ifdef CONFIG_NET_CLS_ROUTE
    __u32         r_tclassid;
#endif
```

This field, r_ifname, is the name of the network interface device.

```
    char          r_ifname[IFNAMSIZ];
    int           r_dead;
} ;
```

There are three fib_rule instances to select each of the "permanent" FIB tables: the main table, the local table, and the default table if no table is specified. As new FIB rules are defined, they will be added to the end of the list.

This is the default table, number 253.

```
static struct fib_rule default_rule = {
    r_clntref:    ATOMIC_INIT(2),
    r_preference: 0x7FFF,
    r_table:      RT_TABLE_DEFAULT,
    r_action:     RTN_UNICAST,
} ;
```

This is the main table, 254.

```
static struct fib_rule main_rule = {
    r_next:       &default_rule,
```

```
    r_clntref:    ATOMIC_INIT(2),
    r_preference: 0x7FFE,
    r_table:      RT_TABLE_MAIN,
    r_action:     RTN_UNICAST,
} ;
```

This is the local table, 255.

```
static struct fib_rule local_rule = {
    r_next:       &main_rule,
    r_clntref:    ATOMIC_INIT(2),
    r_table:      RT_TABLE_LOCAL,
    r_action:     RTN_UNICAST,
} ;
static struct fib_rule *fib_rules = &local_rule;
```

Next, we examine some of the functions that are used to enter, delete, and look up rules in the FIB rules database. Many of these functions are called from the rtnetlink protocol, which passes messages to us from user space in the nlmsghdr structure. The nlmsghdr structure is followed in memory by the rtmsg structure, which contains most of the information used to determine what rule to use. Each of the following functions use fields in the rtmsg structure, which is explained in Section 9.6.5.

The first function we will look at, inet_rtm_delrule, is declared in file *linux/include/net/ip_fib.h* and defined in the file *linux/net/ipv4/fib_rules.c* called from the rtnetlink protocol to delete a rule from the FIB rules database.

```
int inet_rtm_delrule(struct sk_buff *skb, struct nlmsghdr* nlh, void *arg);
```

It searches for a rule to delete in fib_rules by looking at the rtm_type field of the structure pointed to by nlh. The values for rtm_type are shown in Table 9.1. The criteria for the match is based on the source and destination address lengths, rtm_src_len, rtm_dst_len, which can determine the specificity of the route. rtm_tos and the interface name are used to match the rule.

The next function, inet_rtm_newrule, adds a new rule to the FIB rules.

```
int inet_rtm_newrule(struct sk_buff *skb, struct nlmsghdr* nlh, void *arg);
```

It creates a new rule based on the rtm_type in the nlmsghdr. It creates a new FIB rule entry with kmalloc. It is important to note that the rules don't use a slab cache because they don't need to be that fast. These functions are not in the data path, and rules don't get added or deleted often. In this function, we create source and destination netmasks from the rtm_src_len and rtm_dst_len fields, respectively. It gets the r_action from the rtm_type field and the r_flags from the rtm_flags field in nlh. In addition, it gets the network interface device from its name.

The next function, fib_rules_map_destination, returns the Internet gateway address used to reach the destination address, daddr.

```
u32 fib_rules_map_destination(u32 daddr, struct fib_result *res);
```

This function returns the traffic class identifier field of the fib_rules structure, which is attached to the FIB result pointed to by res.

The next function, fib_rules_tclass, is used only when traffic class based routing is configured into the Linux kernel.

```
u32 fib_rules_tclass(struct fib_result *res);
```

The next function, fib_lookup, will be shown in its entirety because it is the primary front-end function to do route searches in the FIB.

```
int fib_lookup(const struct flowi *flp, struct fib_result *res)
{
    int err;
    struct fib_rule *r, *policy;
    struct fib_table *tb;

    u32 daddr = flp->fl4_dst;
    u32 saddr = flp->fl4_src;

FRprintk("Lookup: %u.%u.%u.%u <- %u.%u.%u.%u ",
    NIPQUAD(key->dst), NIPQUAD(flp->fl4_src));
    read_lock(&fib_rules_lock);
```

We search fib_rules looking for a rule that most closely matches the source, destination address, ToS (if ToS routing is configured), and network interface device. Our goal is to get the correct table to search for the route.

```
    for (r = fib_rules; r; r=r->r_next) {
        if (((saddr^r->r_src) & r->r_srcmask) ||
            ((daddr^r->r_dst) & r->r_dstmask) ||
#ifdef CONFIG_IP_ROUTE_TOS
            (r->r_tos && r->r_tos != key->tos) ||
#endif
#ifdef CONFIG_IP_ROUTE_FWMARK
            (r->r_fwmark && r->r_fwmark != key->fwmark) ||
#endif
            (r->r_ifindex && r->r_ifindex != key->iif))
                continue;
```

Now that we have a matching rule, we check the route type. If it is RTN_UNICAST or RTN_NAT, we have found a rule. If not, we return an error.

```
FRprintk("tb %d r %d ", r->r_table, r->r_action);
        switch (r->r_action) {
        case RTN_UNICAST:
        case RTN_NAT:
            policy = r;
            break;
        case RTN_UNREACHABLE:
            read_unlock(&fib_rules_lock);
            return -ENETUNREACH;
        default:
        case RTN_BLACKHOLE:
```

```
            read_unlock(&fib_rules_lock);
            return -EINVAL;
        case RTN_PROHIBIT:
            read_unlock(&fib_rules_lock);
            return -EACCES;
        }
```

Now we get the FIB table for matching rule, r. If we successfully got a table, we call the FIB table's lookup function, which returns the FIB result in res.

```
        if ((tb = fib_get_table(r->r_table)) == NULL)
            continue;
        err = tb->tb_lookup(tb, flp, res);
```

If we were looking for a NAT route, policy will point to the NAT rule, so we change res to point to the NAT FIB rule. This is how the address is remapped, because the FIB rule pointed to by r will contain the mapped address.

```
        if (err == 0) {
            res->r = policy;
            if (policy)
                atomic_inc(&policy->r_clntref);
            read_unlock(&fib_rules_lock);
            return 0;
        }
        if (err < 0 && err != -EAGAIN) {
            read_unlock(&fib_rules_lock);
            return err;
        }
    }
FRprintk("FAILURE\ n");
    read_unlock(&fib_rules_lock);
    return -ENETUNREACH;
}
```

The next function, fib_select_default, selects the default route.

```
void fib_select_default(const struct flowi *flp, struct fib_result *res);
```

This function actually calls tb_select_default through the function pointer in fib_table. First, however, it gets the FIB rule to check a few things. It makes sure that the route type is RTN_UNICAST, that there is a next hop (gateway) associated with the fib_result in res, and that the gateway is directly connected. It knows that the gateway is connected if the nh_scope field in the next-hop structure is RT_SCOPE_LINK. If all of these conditions are met, it calls tb_select_default; otherwise, it returns.

9.6.3 FIB Initialization

The initialization of the FIB for IPv4 is done by the function ip_fib_init, defined in file *linux/net/ipv4/fib_frontend.c*.

```
void __init ip_fib_init(void)
```

```
{
```

If we are configured for multiple tables, we call fib_rules_init to set up the rules part of the RPDB. Otherwise, we call fib_hash_init directly for each of the two tables that are globally defined.

```
#ifndef CONFIG_IP_MULTIPLE_TABLES
    local_table = fib_hash_init(RT_TABLE_LOCAL);
    main_table = fib_hash_init(RT_TABLE_MAIN);
#else
    fib_rules_init();
#endif
```

Next, we register ourselves with the notifier facility in net device. We do this to put ourselves on the chain of functions called when network interface devices change status. Now we can be sure we are informed when we have to add or delete routes from the FIB as devices go up or down. See Chapter 4, "Linux Networking Interfaces and Device Drivers," for more information on the notifier facility.

```
    register_netdevice_notifier(&fib_netdev_notifier);
    register_inetaddr_notifier(&fib_inetaddr_notifier);
}
```

The initialization of a specific FIB table is done by the function fib_hash_init, defined in file *linux/net/ipv4/fib_hash.c*. We have two interfaces to this function depending on whether multiple tables are defined.

```
#ifdef CONFIG_IP_MULTIPLE_TABLES
struct fib_table * fib_hash_init(int id)
#else
struct fib_table * __init fib_hash_init(int id)
#endif
{
```

This variable, tb, will point to the new FIB hash table.

```
    struct fib_table *tb;
```

The first thing we do is create the FIB slab cache. This is the slab cache from which FIB entries, in fib_node structures, will be allocated. The new fib_table is allocated by calling kmalloc. We don't need a slab cache for the fib_table because each system will only have a few routing tables. There are at least two instances of the fib_table structure, RT_TABLE_MAIN and RT_TABLE_LOCAL. If the kernel option, CONFIG_IP_MULTIPLE_TABLES, is configured, there are more fib_table instances.

```
    if (fn_hash_kmem == NULL)
        fn_hash_kmem = kmem_cache_create("ip_fib_hash",
                                         sizeof(struct fib_node),
                                         0, SLAB_HWCACHE_ALIGN,
                                         NULL, NULL);

    tb = kmalloc(sizeof(struct fib_table) + sizeof(struct fn_hash),
```

```
                GFP_KERNEL);
    if (tb == NULL)
        return NULL;
```

Now we initialize the fields of the fib_table. First, the tb_id field is set to the FIB table number. Then, the operation functions are set.

```
    tb->tb_id = id;
```

The following six operation functions are discussed elsewhere in this chapter.

```
    tb->tb_lookup = fn_hash_lookup;
    tb->tb_insert = fn_hash_insert;
    tb->tb_delete = fn_hash_delete;
    tb->tb_flush = fn_hash_flush;
    tb->tb_select_default = fn_hash_select_default;
    tb->tb_dump = fn_hash_dump;
```

The information in each FIB table entry is kept in a structure called a fib_node. The field tb_data points to an array of FIB node hash structures, which in turn contain pointers to fib_node.

```
    memset(tb->tb_data, 0, sizeof(struct fn_hash));
    return tb;
}
```

9.6.4 The FIB to Application Interface

Several application programs handle routing information. Examples of application programs that need to access and set routes include the route(8) program and routing daemons such as zebra or gated. To aid these application programs, Linux provides an interface to the internal kernel routing information stored in the FIB. Information about the routing tables must be passed back and forth between the kernel and application programs. Generally, in Linux, when an application wants to add or delete a route, it will use the rtmsg structure and pass it through a rtnetlink socket. The rtmsg structure was discussed earlier in this chapter, and the rtnetlink sockets were discussed in Chapter 5, "Linux Sockets."

The structure rtentry, defined in file *route.h*, is another way provided to the user interface to retrieve and enter routes into the FIB. This structure is not actually used internally in the FIB. It is similar to the rtentry structure passed in the route ioctls used with the BSD operating systems and helps with application portability between Linux and other flavors of Unix. In this section, we will cover rtentry, which is one way routing information is exchanged. A pointer to the rtentry structure is passed as an argument by ioctl calls such as SIOCADDRT and SIOCDELRT, which add and delete routing information.

```
struct rtentry
{
    unsigned long       rt_pad1;
```

Rt_dst holds the target address, rt_gateway is the gateway address, and rt_genmask is the target netmask.

```
    struct sockaddr     rt_dst;
    struct sockaddr     rt_gateway;
    struct sockaddr     rt_genmask;
```

Rt_flags holds the routing table entry flags. The values for the flags are shown in Table 9.5.

```
    unsigned short      rt_flags;
    short               rt_pad2;
    unsigned long       rt_pad3;
    void                *rt_pad4;
```

Table 9.5: Routing Table Entry Flags

Name	Value	Purpose
RTF_UP	0x0001	The route is usable.
RTF_GATEWAY	0x0002	The destination is a gateway.
RTF_HOST	0x0004	This route is a host entry, not a net route.
RTF_REINSTATE	0x0008	Flag says to reinstate route after it ages out.
RTF_DYNAMIC	0x0010	This route is created dynamically by a redirect.
RTF_MODIFIED	0x0020	Route is modified dynamically by a redirect.
RTF_MTU	0x0040	There is a specific MTU for this route.
RTF_MSS	RTF_MTU	Route maximum segment size. It is the same as RTF_MTU and is provided for compatibility with the BSD OS.
RTF_WINDOW	0x0080	This indicates that there is window clamping for this route.
RTF_IRTT	0x0100	Initial round-trip time.
RTF_REJECT	0x0200	Says to reject route.

The next field, rt_metric, is used to pass in the routing metric or priority.

```
    short               rt_metric;
```

The user passes a pointer to a device name in rt_dev if he wants to add a route for a specific device.

```
    char                *rt_dev;
```

This field, rt_mtu, and its equivalent, rt_mss, are set by the application to pass the MTU value to specify a maximum packet or segment size for a specific route.

```
    unsigned long       rt_mtu;
#ifndef __KERNEL__
#define rt_mss    rt_mtu
#endif
```

The following two fields can be used to maintain two specific parameters for TCP, the TCP window size clamp and the initial round-trip time.

```
    unsigned long       rt_window;
```

```
    unsigned short      rt_irtt;
} ;
```

The main method of accessing the FIB is through netlink sockets. Netlink sockets provide extensions for routing called routing netlink sockets, or rtnetlink. The netlink mechanism is explained in Chapter 5. Here, however, we will show the functions rtnetlink uses internally to add or delete entries in the FIB. For example, the following two functions add and delete routes from the FIB. They act as front ends to the internal FIB functions accessed through the tb_insert and tb_delete function pointers in the fib_table structure. These functions are declared in file *linux/include/net/ip_fib.h* and defined in *linux/net/ipv4/fib_frontend.c*. Both functions accept a nlmsghdr structure as an argument because they are called from netlink recvmsg and sendmsg functions. The nlmsghdr has an associated rtmsg structure, which was shown earlier.

The function inet_rtm_newroute adds a new route to the FIB.

```
int inet_rtm_newroute(struct sk_buff *skb, struct nlmsghdr* nlh, void *arg);
```

The next function, inet_rtm_delroute, deletes a route from the FIB.

```
int inet_rtm_delroute(struct sk_buff *skb, struct nlmsghdr* nlh, void *arg)
```

Both of these functions extract the rtmsg structure, which is placed in memory immediately after the nlmsg header pointed to by nlh. The rtmsg structure is defined in file *linux/include/linux/rtnetlink.h*.

```
struct rtmsg
{
    unsigned char       rtm_family;
```

The following two fields are the number of bits to use to create a 32-bit or smaller netmask for AF_INET type addresses for both the source and destination.

```
    unsigned char       rtm_dst_len;
    unsigned char       rtm_src_len;
```

This field corresponds to the ToS field in the IP header.

```
    unsigned char       rtm_tos;
```

Rtm_table contains the routing table ID. If multiple tables are not configured, this would be either RT_TABLE_MAIN or RT_TABLE_LOCAL.

```
    unsigned char       rtm_table;
```

Rtm_protocol refers to the routing message protocol from Table 9.6. The next field, rtm_scope, is the route message scope and the values for this field are the same as those listed earlier in Table 9.2.

```
    unsigned char       rtm_protocol;
    unsigned char       rtm_scope;
```

Table 9.6: Route Message Protocol Field Values

Protocol	Value	Purpose
RTPROT_UNSPEC	0	The value is unspecified.
RTPROT_REDIRECT	1	This route is installed by ICMP redirects. This is not used currently by IPv4.
RTPROT_KERNEL	2	This route is installed by kernel.
RTPROT_BOOT	3	The route is installed during boot.
RTPROT_STATIC	4	Route is installed by the administrator.
RTPROT_GATED	8	Used by the gated routing daemon
RTPROT_RA	9	Used for RDISC/ND router advertisements.
RTPROT_MRT	10	This value is used by Merit MRT.
RTPROT_ZEBRA	11	Used by Zebra routing demon.
RTPROT_BIRD	12	Used by BIRD.
RTPROT_DNROUTED	13	Used by DECnet routing daemon.

This field is the routing message type, rtm_type. It has the same values as the route types shown in Table 9.1.

```
    unsigned char       rtm_type;
```

The next field, rtm_flags, can be one of three values; RTM_F_NOTIFY or 0x100 says to notify the user of route change. RTM_F_CLONED, or 0x200 indicates that this route is cloned, and the final value, RTM_F_EQUALIZE, or 0x400 is not implemented yet but is for a multipath equalizer.

```
    unsigned            rtm_flags;
} ;
```

The values for the protocol field of the rtmsg structure are shown in Table 9.4. The values greater than RTPROT_STATIC are not interpreted by the kernel and are passed between user space and the kernel without modification. They are intended to be used by hypothetical multiple routing daemons. The comments in the code recommend that the values should be standardized to avoid conflicts.

The next function handles the IP routing ioctl call, ip_rt_ioctl, declared in file *linux/include/net/route.h* and defined in *linux/net/ipv4/fib_frontend.c*. This function is another way for an application to add or delete routes from the routing tables. These functions are provided for compatibility to some older legacy applications. However, most applications that want to exchange routing information with the internal routing table will use the /proc file system proc(5), or the sysctl interface, sysctl(2), or netlink sockets, netlink(4).

```
int ip_rt_ioctl(unsigned int cmd, void *arg);
```

The two ioctls supported by ip_rt_ioctl are SIOCADDRT and SIOCDELRT to add and delete a route, respectively. The routing control flags are shown in Table 9.7.

Table 9.7: IPv4 Routing Control Flags

Flag	Value	Purpose
RTCF_DEAD	RTNH_F_DEAD	Dead route.
RTCF_ONLINK	RTNH_F_ONLINK	Indicates a locally reachable destination.
RTCF_NOPMTUDISC	RTM_F_NOPMTUDISC	This is an obsolete flag.
RTCF_NOTIFY	0x00010000	Indicates (in theory) that a notification should be sent every time this route becomes stale.
RTCF_DIRECTDST	0x00020000	The destination is directly reachable. This flag does not appear to be used.
RTCF_REDIRECTED	0x00040000	The source address for this route is directly reachable; it is a local route.
RTCF_TPROXY	0x00080000	Not used.
RTCF_FAST	0x00200000	Route has not been redirected (if fast routing is enabled).
RTCF_MASQ	0x00400000	Indicates that the source address is masqueraded.
RTCF_SNAT	0x00800000	Indicates that the source address in this route is actually a translated source address.
RTCF_DOREDIRECT	0x01000000	Generate ICMP redirect.
RTCF_DIRECTSRC	0x04000000	Destination is directly connected.
RTCF_DNAT	0x08000000	The destination address is translated.
RTCF_BROADCAST	0x10000000	Broadcast route.
RTCF_MULTICAST	0x20000000	Multicast route.
RTCF_REJECT	0x40000000	This flag does not appear to be used.
RTCF_LOCAL	0x80000000	Local route.
RTCF_NAT	(RTCF_DNAT| RTCF_SNAT)	Indicates that either the destination or the source for this route is a translated address.

9.6.5 FIB Internal Kernel Interface

Many internal functions have to access routing information from the FIB. When the routing mechanism needs to determine the route for a packet, it first uses a hash key to search the routing table whose entries are defined by the rtable structure, shown earlier. Once routes are known, they are cached in the routing table cache. Therefore, the FIB provides several functions to convert to and from the rtable entry and the internal FIB representation.

Another data structure is sometimes used to represent routing table entries when they are added or deleted by internal modules. This is the kern_rta structure, defined in file *linux/include/ip_fib.h*.

```
struct kern_rta
{
    void          *rta_dst;
    void          *rta_src;
    int           *rta_iif;
    int           *rta_oif;
    void          *rta_gw;
    u32           *rta_priority;
    void          *rta_prefsrc;
    struct rtattr *rta_mx;
    struct rtattr *rta_mp;
    unsigned char *rta_protoinfo;
    unsigned char       *rta_flow;
    struct rta_cacheinfo       *rta_ci;
    struct rta_session        *rta_sess;
} ;
```

Another data structure used to return the result of a lookup in the FIB. We saw earlier that a pointer to this structure is returned by the fib_lookup function. This data structure is fib_result, defined in file *linux/include/net/ip_fib.h*.

```
struct fib_result
{
```

Prefixlen is the prefix length. This value is used to construct the netmask for the route and consists of a number from zero to 32, which indicates how many bits of the address to mask.

```
    unsigned char prefixlen;
    unsigned char nh_sel;
```

Type could be thought of as the type of the route contained in this entry. It actually can be converted to the flags for the routing table entry, which are defined in Table 9.5. Scope is the scope of the route.

```
    unsigned char type;
    unsigned char scope;
```

The fib_info structure is shown earlier in this section.

```
    struct fib_info     *fi;
```

If multiple routing tables are configured, the following field points to the fib_rule structure used to get this result.

```
#ifdef CONFIG_IP_MULTIPLE_TABLES
    struct fib_rule     *r;
#endif
} ;
```

Previously, we introduced the data structures used internally by the FIB and the data structures used to interface to the FIB. Now we describe some of the functions that are commonly used to interface to the FIB. Most of these functions are defined in *fib_frontend.c*. However, we will explain some of the FIB utility functions that are accessed through the function pointers in the fib_table structure. The FIB is a generic resource to all protocols, not just IPv4; therefore, it is protocol independent. The external interfaces have no dependency on IPv4 32-bit address formats.

The first function we will discuss is __fib_new_table, which creates a new FIB table. This function is defined in the file *linux/net/ipv4/fiib_frontend.c*. Id contains the new table ID, and it must be between 1 and 252. The higher table numbers are used by the permanent FIB tables.

```
struct fib_table *__fib_new_table(int id);
```

It calls fib_get_table to retrieve the FIB table, id.

The next function, fib_flush, declared in the file *linux/include/net/ip_fib.h*, removes all entries from all of the FIB tables.

```
void fib_flush(void);
```

If multiple tables are configured into the kernel, fib_flush gets all the tables for the full range of FIB IDs from one to RT_TABLE_MAX. It calls the flush operation function for each table. When it is done with the FIB tables, it flushes the route cache. Even if multiple tables are not configured, we will always have two tables defined, the main table and the local table. Both tables are defined as global variables in *linux/net/ipv4/fib_frontend.c*.

Ip_dev_find returns the output network interface that has been configured with the address, addr. It is declared in *linux/include/linux/inetdevice.h* and im plemented in *linux/net/ipv4/fib_frontend.c*. It returns a pointer to the net_device structure for the device. It is interesting to examine this structure because it shows how the local table in the FIB is used to keep local address information.

```
struct net_device * ip_dev_find(u32 addr);
{
```

First, we build the flowi structure to do the search of the FIB. The only field in flowi that we use is the destination address.

```
    struct flowi fl = {  .nl_u = {  .ip4_u = {  .daddr = addr }  }  } ;
    struct fib_result res;
    struct net_device *dev = NULL;
#ifdef CONFIG_IP_MULTIPLE_TABLES
    res.r = NULL;
#endif
```

It does the lookup in the local FIB table, local_table, which includes all IP address information within this machine. Each FIB table has its own set of service functions. In this example, the lookup routine is called through the tb_lookup function field of the fib_table structure for the

local table. The returned routes must be of type RTN_LOCAL, or there is a bogus entry in the table.

```
    if (!local_table || local_table->tb_lookup (local_table, &key, &res)) {
        return NULL;
    }
    if (res.type != RTN_LOCAL)
        goto out;
    dev = FIB_RES_DEV(res);
```

Once we have a reference to the network interface device, we must increment the use count in the net_device structure.

```
    if (dev)
        atomic_inc(&dev->refcnt);
out:
    fib_res_put(&res);
    return dev;
}
```

The function inet_addr_type, defined in file *linux/include/net/route.h*, looks up the address, addr, in the local FIB table and returns an address type of unicast, broadcast, or multicast. The return values for this function are shown in Table 9.1.

```
unsigned inet_addr_type(u32 addr);
```

The following function, fib_validate_source, declared in file *linux/include/net/ip_fib.h* and implemented in *linux/net/ipv4/fib_frontend.c*, provides a way of verifying the source address. A side effect of this function is that it retrieves the specific destination address and sets spec_dst to point to the address The specific destination address is used as the source address for outgoing UDP packets from multihomed hosts. The function also gets the route tag or traffic class identifier, which is used for traffic shaping. The function sets the parameter, itag, to point to the traffic class identifier for the next hop to the destination address.

```
int fib_validate_source(u32 src, u32 dst, u8 tos, int oif,
                        struct net_device *dev, u32 *spec_dst, u32 *itag)
{
```

This function will be called to make sure that there is a route to a source address. For example, if the caller needs to verify the source address in an input packet, it can call fib_validate_source to check that the source address is valid.

```
    struct in_device *in_dev;
```

Flowi is initialized with the information for the FIB table lookup. We set fields in flowi based on the arguments, but some of the fields are optional. The only required parameter for a meaningful lookup is the source address, saddr.

```
    struct flowi fl = {  .nl_u = {  .ip4_u =
                        {  .daddr = src,
                          .saddr = dst,
```

```
                        .tos = tos }  } ,
                        .iif = oif } ;
    struct fib_result res;
    int no_addr, rpf;
    int ret;
```

We get the inet_device structure for the interface, dev. If no_addr is zero, it indicates that the interface does not have an IP address. If rpf is one, then the reverse path filtering option is enabled for the interface, dev.

```
    no_addr = rpf = 0;
    read_lock(&inetdev_lock);
    in_dev = __in_dev_get(dev);
    if (in_dev) {
        no_addr = in_dev->ifa_list == NULL;
        rpf = IN_DEV_RPFILTER(in_dev);
    }
    read_unlock(&inetdev_lock);

    if (in_dev == NULL)
        goto e_inval;
```

Now we call fib_lookup to search the FIB. This function returns zero if the lookup succeeds. If the route type is not unicast, then the source address para meter, src, must be Martian because multicast or broadcast source addresses don't make any sense.

```
    if (fib_lookup(&fl, &res))
        goto last_resort;
    if (res.type != RTN_UNICAST)
        goto e_inval_res;
```

We get the "specific destination address" from the fib_info structure attached to the FIB result, res. The special destination address is used by UDP as the source address for response packets.

```
    *spec_dst = FIB_RES_PREFSRC(res);
```

Fib_combine_itag generates the traffic class tag. In the simplest case, it returns the traffic class identifier of the next hop. If multiple tables are configured and we are doing traffic shaping, it returns the traffic class ID in the fib_rules for the route.

```
    fib_combine_itag(itag, &res);
```

This is where we check of the validity of the output device. Dev must be the same as the device shown in route entry (unless multipath routing is configured). In other words, the network interface at which the packet arrived should be the same as the network device for the route to the host from which the packet was sent.

```
#ifdef CONFIG_IP_ROUTE_MULTIPATH
    if (FIB_RES_DEV(res) == dev || res.fi->fib_nhs > 1)
#else
    if (FIB_RES_DEV(res) == dev)
#endif
```

```
    {
        ret = FIB_RES_NH(res).nh_scope >= RT_SCOPE_HOST;
        fib_res_put(&res);
        return ret;
    }
    fib_res_put(&res);
    if (no_addr)
        goto last_resort;
    if (rpf)
        goto e_inval;
    key.oif = dev->ifindex;
```

If the test for a network interface failed, we might still have some type of logical device. We do another lookup, and if this lookup is successful, we recalculate the special destination address.

```
    ret = 0;
    if (fib_lookup(&fl, &res) == 0) {
        if (res.type == RTN_UNICAST) {
            *spec_dst = FIB_RES_PREFSRC(res);
            ret = FIB_RES_NH(res).nh_scope >= RT_SCOPE_HOST;
        }
        fib_res_put(&res);
    }
    return ret;
```

We end up here if the FIB lookup failed, so we call inet_select_addr to find the specific distribution address. This function will return the address assigned to network interface, dev, to be used for the special destination address.

```
last_resort:
    if (rpf)
        goto e_inval;
    *spec_dst = inet_select_addr(dev, 0, RT_SCOPE_UNIVERSE);
    *itag = 0;
    return 0;

e_inval_res:
    fib_res_put(&res);
e_inval:
    return -EINVAL;
}
```

9.6.6 FIB Table Hash Functions

Under the covers, the FIB table is implemented as a multizone hash table. Each location in the hash table is held by a fib_info structure. The fib_info structure is explained in Section 9.6.1. The fib_info structures are allocated from a slab cache called the fib node cache, which is also defined in file *linux/net/ipv4/fib_hash.c*.

```
static kmem_cache_t * fn_hash_kmem;
```

Next, we will look at the hash functions defined for the fib_table. As we know from earlier discussions, the FIB is a generic routing database. The fib_table structure consists of a set of

pointers to operation functions. Actions affecting the FIB are done through these function pointers, which could be over-ridden by specific protocol implementations of the FIB. The functions for the IPv4 FIB implementation are defined in *linux/net/ipv4/fib_hash.c*. Each of the functions performs multizone hash calculations to locate particular FIB entries. The functions that retrieve entries from the FIB store the result in a feb_result structure pointed to by the argument, res. Functions that add entries to the FIB accept a pointer to the kern_rta structure as an argument. Most of the functions return a zero on success and a negative error if unsuccessful.

The first function, fn_hash_lookup, also defined in file *fib_hash.c*, is the low-level function provided to look up a route entry in the FIB.

```
static int fn_hash_lookup(struct fib_table *tb, const struct flowi
*fl, struct fib_result *res);
```

It accepts a pointer to a FIB table, tb, and routing flow information in fl and returns the result of the search in the fib_result structure pointed to by res. In this function, we retrieve the FIB node hash table from the field, tb_data in fib_table. Before doing the lookup, we form a FIB zone key from the destination address in the IPv4 part of the flowi structure, fl4_dst. If we get a match on the address, we set the type, scope, and prefixlen in the fib_result structure pointed to by res.

The next function, fn_hash_insert, adds a new route to the FIB table, tb.

```
static int fn_hash_insert(struct fib_table *tb, struct rtmsg *r, struct
kern_rta *rta, struct nlmsghdr *n, struct netlink_skb_parms *req);
```

It accepts information about the route to be added in the rtmsg structure because it is called from the rtnetlink protocol. Information about the result is returned in the kern_rta structure rta.

The next function, fn_hash_delete, deletes an entry from the FIB table, tb.

```
static int fn_hash_delete(struct fib_table *tb, struct rtmsg *r, struct
kern_rta *rta, struct nlmsghdr *n, struct netlink_skb_parms *req);
```

The next function, fn_hash_flush, is much simpler.

```
static int fn_hash_flush(struct fib_table *tb);
```

It removes all entries in the FIB table, tb. It returns the number of entries it found or zero if it didn’t find any.

Fn_hash_dump dumps the contents of the FIB table, tb.

```
static int fn_hash_dump(struct fib_table *tb, struct sk_buff *skb,
struct netlink_callback *cb);
```

It calls the callback function cb for each entry in the FIB table.

This function, fn_hash_select_default, is called as the lowest-level function to find a default route in the FIB table, tb.

```
static void fn_hash_select_default(struct fib_table *tb, const struct
rt_key *key, struct fib_result *res);
```

This function is usually called from internal kernel routines such as fib_select_default and not called directly from rtnetlink. It searches for the route indicated by the rt_key structure key. It returns the result in the fib_result structure pointed to by res.

9.7 Routing Input Packets

As we will show later in this chapter, the IP receive packet handler makes a decision as to how to route each incoming packet. TCP requests a route for input packets when the accept socket call is issued. There are two stages to this decision process, *fast path routing* and *slow path routing*. The fast path consists simply of a lookup in the routing cache. The fast path looks for a match on the incoming interface, the source, and the destination IP addresses to see if there is a cached route that applies to this packet. The slow path consists of a more time-consuming search of the FIB for the most appropriate applicable match. First, we will discuss the fast path routing in this section. A later section will cover slow path routing.

9.7.1 Routing Input Packets—the Fast Path

Each packet has a destination that at any given time may either be known or unknown. As a packet is transmitted, once the destination is known, the destination cache entry, dst, in the socket buffer structure, sk_buff, is used to quickly extract the link layer header including the hardware destination address. The purpose of fast path routing is to find the entry in the route cache and place a pointer to it in the socket buffer. When we are done, the dst field in the socket buffer is a pointer to an entry in the destination cache for a specific route.

For reasons of efficiency, internally we handle incoming packets the same way as outgoing packets. Therefore, we also consider incoming packets as having a destination. Generally, this destination may be a transport protocol or it may be the packet forwarding routine. In this section, we examine how the destination cache entry is obtained by searching the route cache with a key based on a few fields of the incoming packet. The destination of the incoming packet is in the IP forwarding routine, and if that is the case, IP forward will worry about how to extract the link layer header from the destination cache when the packet is ready to be transmitted.

Routing information for an incoming packet is obtained during input packet processing. The IP receive routine, after it does some basic validity checking of the incoming IP packet, must decide where to send the packet next. To make this decision, it calls ip_route_input to get the destination for an incoming packet. Ip_route_input, defined in file *linux/net/ipv4/route.c*, gets a route to the destination by first searching for the route table entry in the routing table cache. If there is no matching entry in the cache, it calls ip_route_input_slow to search the FIB. Ip_route_input returns a zero if a route was found.

```
int ip_route_input(struct sk_buff *skb, u32 daddr, u32 saddr,
                   u8 tos, struct net_device *dev)
{
```

Rth is a pointer to a route table entry. Hash is the first-level hash code to find a location in the rt_hash_table. We are trying to route an input packet, so we set iif to the incoming network interface index.

```
    struct rtable * rth;
    unsigned      hash;
    int iif = dev->ifindex;
    tos &= IPTOS_RT_MASK;
```

We call rt_hash_code to calculate a 4-byte hash code from the source and destination addresses, the type of service field, and the input interface in the incoming packet, skb.

```
    hash = rt_hash_code(daddr, saddr ^ (iif << 5), tos);
```

Each location in the hash bucket array, rt_hash_table, has a read/write lock. As a reader of the table, we read-lock the location before searching the chain. Anyone writing to the table must write-lock the same location because a use count is incremented once a route table entry is used.

```
    read_lock(&rt_hash_table[hash].lock);
```

We use the hash code, hash, to start our search at a location in the hash bucket table. In this location, the field chain points to a linked list of routes. An item on the list can be an entry in the destination cache or a route table entry.

```
    for (rth = rt_hash_table[hash].chain; rth; rth = rth->u.rt_next) {
```

Now we follow the chain of routes to find a precise match by comparing the destination and source addresses, the input interface index, and the IP ToS field

```
        if (rth->fl.fl4_dst == daddr &&
            rth->fl.fl4_src == saddr &&
            rth->fl.iif == iif &&
            rth->fl.oif == 0 &&
#ifdef CONFIG_IP_ROUTE_FWMARK
            rth->fl.fl4_fwmark == skb->nfmark &&
#endif
            rth->fl.fl4_tos == tos) {
```

If we have a match, we are actually in the fast path for input packets. This means that the route has been resolved and the matching route entry actually points to a destination cache entry, not a route table entry. Both structures begin with a next field so they can be overloaded in the same cache. The dst_entry structure is shown later in this section, and a more detailed explanation of the destination entry cache can be found in Chapter 6.

We have a match, and we know that rth actually points to the destination cache. The destination cache is defined by the structure dst_entry, We can access this structure through the union, u, in the beginning of the rtable structure. We update a few fields in the dst_entry structure now that we have a hit. We mark the time when this destination cache entry was last used by setting lastuse to the current time. (The global kernel variable, jiffies, holds the current time in ticks.)

```
            rth->u.dst.lastuse = jiffies;
```

We increment the reference count, __refcnt by, calling dst_hold. The use count, __use, is incremented also. Cache statistics are updated to indicate that we have a routing cache hit. Finally, the dst field of the socket buffer, skb, is updated to point to rth, which is actually a destination cache entry. We return zero to in dicate that we have found a route for this packet. Before returning, we unlock the location in the hash bucket, rt_hash_table.

```
            dst_hold(&rth->u.dst);
            rth->u.dst.__use++;
            RT_CACHE_STAT_INC(in_hit);
            read_unlock(&rt_hash_table[hash].lock);
            skb->dst = (struct dst_entry*)rth;
            return 0;
        }
        RT_CACHE_STAT_INC(in_hlist_search);
    }
```

If we end up here, it means that we did not find a matching route in the route table cache. We could be in the slow path for unicast routes or we may have a multicast destination address. First, we unlock the location in the hash bucket, rt_hash_table, because we are done with the routing cache.

```
    rcu_read_unlock();
```

We check to see if the destination IP address, daddr, is a multicast address. The comments in the code indicate that multicast recognition in earlier releases was done in the route cache. However, some types of Ethernet interfaces didn't properly recognize hardware multicast addresses, so the number of entries in the routing cache could explode. To avoid this problem, host multicast address decoding is done here outside the routing cache. Multicast routers will still use the route cache.

```
    if (MULTICAST(daddr)) {
```

The first thing we do if daddr is a multicast destination address is get the Internet device structure, in_device, from the network interface device structure, dev. In_device structure contains the multicast address list among other things. The in_device structure has a global read/write spin lock so we read lock the structure to protect it.

```
        struct in_device *in_dev;
        read_lock(&inetdev_lock);
        if ((in_dev = __in_dev_get(dev)) != NULL) {
```

We call ip_check_mc to see if the destination address of the incoming packet, daddr, is one of the addresses on the list of multicast addresses for the incoming network interface. If so, our is set to TRUE. This means that the input interface has been added to the multicast group for the address, daddr.

```
            int our = ip_check_mc(in_dev, daddr, saddr,
                                  skb->nh.iph->protocol);
```

Ip_route_input_mc is called to route input multicast packets if a couple of specific conditions are met. It is called if our is not zero, indicating that the incoming network interface has subscribed to the multicast address, daddr. Next, we call ip_route_input_mc if multicast routing is configured in the kernel and the multicast forwarding option is set for the input interface. However, if daddr is a multicast address but none of these other conditions are met, we return an error indicating that the input packet, skb, should be dropped.

```
            if (our
#ifdef CONFIG_IP_MROUTE
                || (!LOCAL_MCAST(daddr) && IN_DEV_MFORWARD(in_dev))
#endif
                    ) {
                read_unlock(&inetdev_lock);
                return ip_route_input_mc(skb, daddr, saddr,
                                         tos, dev, our);
            }
        }
        read_unlock(&inetdev_lock);
        return -EINVAL;
    }
```

This is the slow path and we arrive here if there was no routing cache hit for the incoming packet, skb, and daddr is not a multicast address.

```
    return ip_route_input_slow(skb, daddr, saddr, tos, dev);
}
```

The destination entry cache structure, dst_entry, is explained in Chapter 6. This structure is referenced by the ip_route_input structure in the case where there is a routing table cache hit. Refer to Chapter 6 for more information about the destination cache.

9.7.2 The Main Input Route Resolving Function

In the previous section, we discussed how the routing table cache is accessed for fast path routing of incoming packets. If there is no cache hit in the attempt to route the packet in the fast path, we try to find a route for the incoming packet by accessing the FIB. This is called the slow path.

The function that tries to find a slow path route for the incoming packet is ip_route_input_slow. This function is called from ip_route_input if there is no routing cache hit for the incoming packet, skb.

```
int ip_route_input_slow(struct sk_buff *skb, u32 daddr, u32 saddr,
                        u8 tos, struct net_device *dev)
{
```

Res points to the result of the FIB search. The fib_result structure is described earlier in this chapter.

```
    struct fib_result res;
```

In_dev is the Internet device associated with the incoming network interface, dev. This structure was also used in the fast path routing described in the previous section. Since we are routing incoming packets, out_dev is NULL.

```
    struct in_device *in_dev = in_dev_get(dev);
    struct in_device *out_dev = NULL;
```

Next, we initialize the flow structure for the FIB search. The fields for addresses and TOS are initialized.

```
    struct flowi fl = {  .nl_u = {  .ip4_u =
                         {  .daddr = daddr,
                         .saddr = saddr,
                         .tos = tos,
```

We set the scope field in the search key to be as broad as possible. The values for route scope are shown in Table 9.2. Later in the function, as we attempt to narrow the routing decision, this value may be narrowed based on the result of the FIB lookup.

```
                             .scope = RT_SCOPE_UNIVERSE,
#ifdef CONFIG_IP_ROUTE_FWMARK
                             .fwmark = skb->nfmark
#endif
                               }  } ,
                             .iif = dev->ifindex } ;
```

The values in flags are the route control flags and they are derived from information in the FIB as we determine the route. They help provide instructions to the consumers of this packet as to how to dispose of the packet. The route control flags are shown in Table 9.7. Itag is used for setting the traffic class, the tclassid field in the destination cache entry, used for traffic shaping.

```
    unsigned      flags = 0;
    u32           itag = 0;
```

Rth points to a routing table cache entry described in an earlier section. Hash is the hash code. Spec_dst is the special destination address.

```
    struct rtable * rth;
    unsigned      hash;
    u32           spec_dst;
    int           err = -EINVAL;
    int           free_res = 0;
```

If in_dev is NULL, there is no AF_INET information for the incoming network interface, dev, and therefore no IP address. IP packets are disabled for this input device.

```
    if (!in_dev)
        goto out;
```

We calculate the hash code for finding entries in the routing table cache.

```
    hash = rt_hash_code(daddr, saddr ^ (key.iif << 5), tos);
```

Some Martian source addresses are filtered out first because they can't be detected by a lookup of the FIB. Afterwards, we see if the incoming packet has been sent to a broadcast address, and if so, we jump to brd_input for further processing.

```
    if (MULTICAST(saddr) || BADCLASS(saddr) || LOOPBACK(saddr))
        goto martian_source;
```

We check if we have a broadcast.

```
    if (daddr == 0xFFFFFFFF || (saddr == 0 && daddr == 0))
        goto brd_input;
```

We filter out zero source addresses. They are only accepted if they also have zero broadcast addresses.

```
    if (ZERONET(saddr))
        goto martian_source;
```

If the destination IP address does not make any sense, we filter it out. We should not see packets with destination loopback addresses at this point.

```
    if (BADCLASS(daddr) || ZERONET(daddr) || LOOPBACK(daddr))
        goto martian_destination;
```

At this point, a fast search of the routing table cache did not yield a hit, but the previous checks indicate that we have valid source and destination unicast addresses. Therefore, we need to find a route for the packet in the FIB, and we do this by calling fib_lookup. Fib_lookup does the search using the flowi structure. It provides the result in a fib_result structure pointed to by res. It returns a zero if the search was successful, or an error if not.

```
    if ((err = fib_lookup(&fl, &res)) != 0) {
        if (!IN_DEV_FORWARD(in_dev))
            goto e_inval;
        goto no_route;
    }
    free_res = 1;
    RT_CACHE_STAT_INC(in_slow_tot);
```

The following section is for Network Address Translation (NAT). If it is configured into the kernel, we attempt to re-map the addresses to the translated addresses.

```
#ifdef CONFIG_IP_ROUTE_NAT
```

If NAT is configured, typically multiple routing tables will be configured. The additional tables contain the address mapping. Here we access the mapping policy in the FIB rules structure pointed to by the result res. The routing policy is applied before re-mapping the destination address. The unmodified source address is used to do re-routing later.

```
    if (1) {
```

```
        u32 src_map = saddr;
```

The field r in fib_result points to the fib_rules data structure. It will be non-NULL if multiple tables are configured.

```
            src_map = fib_rules_policy(saddr, &res, &flags);

        if (res.type == RTN_NAT) {
            key.dst = fib_rules_map_destination(daddr, &res);
            fib_res_put(&res);
            free_res = 0;
            if (fib_lookup(&key, &res))
                goto e_inval;
            free_res = 1;
            if (res.type != RTN_UNICAST)
                goto e_inval;
            flags |= RTCF_DNAT;
        }
        key.src = src_map;
    }
#endif
```

The type of the route is returned in the type field of the fib_result. If the returned route type is broadcast, we complete processing at the brd_input label.

```
    if (res.type == RTN_BROADCAST)
        goto brd_input;
```

The returned route type is RTN_LOCAL if the destination matched one of our addresses.

```
    if (res.type == RTN_LOCAL) {
        int result;
        result = fib_validate_source(saddr, daddr, tos,
                                     loopback_dev.ifindex,
                                     dev, &spec_dst, &itag);
        if (result < 0)
            goto martian_source;
        if (result)
            flags |= RTCF_DIRECTSRC;
```

We know now that we have a local route. The packet should not be forwarded. It should be passed to the higher-level protocols because the packet was sent to one of our local addresses. Spec_dst holds the value for the rt_spec_dst field of rtable. This is the UDP-specific destination address used in certain UDP applications for the source address of an outgoing packet as specified by [RFC 1122]. Finally, we jump to local_input to complete processing.

```
        spec_dst = daddr;
        goto local_input;
    }
```

At this point, we know we will have to forward the incoming packet. It was rejected for the fast route match by ip_route_input, or we wouldn't have ended up here. By failing the previous tests in this function, we know it is not a multicast packet, it is not a broadcast packet, and it is not

being sent to one of our addresses. There are a few other tests before we prepare to route the packet. If the input device is not configured for IP forwarding, we have an error condition. If the route type returned from the FIB is not unicast, we assume there is an error condition; the destination address must have been Martian.

```
    if (!IN_DEV_FORWARD(in_dev))
        goto e_inval;
    if (res.type != RTN_UNICAST)
        goto martian_destination;
```

This section is for multipath routing. It is a check to see if there is more than one gateway machine or next hop defined for this route.

```
#ifdef CONFIG_IP_ROUTE_MULTIPATH
    if (res.fi->fib_nhs > 1 && key.oif == 0)
        fib_select_multipath(&fl, &res);
#endif
```

We get a pointer to the inet_device associated with the output network in terface in the route we got from the FIB. All valid network interface devices that handle IP packets should have an inet_device.

```
    out_dev = in_dev_get(FIB_RES_DEV(res));
    if (out_dev == NULL) {
        if (net_ratelimit())
            printk(KERN_CRIT "Bug in ip_route_input_slow(). "
                   "Please, report\ n");
        goto e_inval;
    }
```

The main reason why we call fib_validate_source here is to calculate the specific destination address, spec_dst. However, the function also helps in making the routing decisions. It returns a number less than zero if the source address is bad, and a number greater than zero if the destination is reachable by direct connection.

```
    err = fib_validate_source(saddr, daddr, tos, FIB_RES_OIF(res), dev,
                              &spec_dst, &itag);
    if (err < 0)
        goto martian_source;
```

We set RTCF_DIRECTSRC in the router control flags to indicate that the destination is directly reachable if fib_validate_source returns a positive value.

```
    if (err)
        flags |= RTCF_DIRECTSRC;
```

Now we check to see if an ICMP redirect error will need to be sent. An ICMP redirect is sent from a router back to the sender to tell it that the packet was sent to the wrong router. In the following test, we are checking to see that the gateway address returned by the RIB_RES_GW macro would be reached through the same interface through which the packet arrived. This is the basic redirect case. In Section 9.5 [STEV94], Stevens has an excellent explanation of ICMP

redirect errors. If this test passes, we set RTCF_DOREDIRECT in the control flags. Later, the ip_forward function will actually generate the ICMP message.

```
    if (out_dev == in_dev && err && !(flags & (RTCF_NAT | RTCF_MASQ)) &&
           (IN_DEV_SHARED_MEDIA(out_dev) ||
           inet_addr_onlink(out_dev, saddr, FIB_RES_GW(res))))
        flags |= RTCF_DOREDIRECT;
```

If the protocol of the incoming packet is not IP, it is not routed.

```
    if (skb->protocol != __constant_htons(ETH_P_IP)) {
        if (out_dev == in_dev && !(flags & RTCF_DNAT))
            goto e_inval;
    }
```

At this point, we know that the incoming packet will be routed. We also know that the destination machine or the gateway is theoretically reachable. Therefore, we allocate a location in the destination cache by calling dst_alloc. Remember that the routing table cache entry can also point to a destination cache entry, dst_entry.

```
    rth = dst_alloc(&ipv4_dst_ops);
    if (!rth)
        goto e_nobufs;
    atomic_set(&rth->u.dst.__refcnt, 1);
```

The flags field of the dst_entry is not used very widely. Other fields in it are set in preparation for entering the route in the route cache.

```
    rth->u.dst.flags= DST_HOST;
    if (in_dev->cnf.no_policy)
        rth->u.dst.flags |= DST_NOPOLICY;
    if (in_dev->cnf.no_xfrm)
        rth->u.dst.flags |= DST_NOXFRM;
    rth->fl.fl4_dst     = daddr;
    rth->rt_dst   = daddr;
    rth->fl.fl4_tos     = tos;
#ifdef CONFIG_IP_ROUTE_FWMARK
    rth->fl.fl4_fwmark  = skb->nfmark;
#endif
    rth->fl.fl4_src     = saddr;
    rth->rt_src  = fl.fl4_dst;
    rth->rt_gateway     = daddr;
#ifdef CONFIG_IP_ROUTE_NAT
    rth->rt_src_map     = fl.fl4_src;
    rth->rt_dst_map     = fl.fl4_dst;
    if (flags&RTCF_DNAT)
        rth->rt_gateway= fl.fl4_dst;
#endif
    rth->rt_iif  =
    rth->fl.iif  = dev->ifindex;
    rth->u.dst.dev      = out_dev->dev;
    dev_hold(rth->u.dst.dev);
    rth->fl.oif  = 0;
    rth->rt_spec_dst= spec_dst;
```

Here, we set the input field of the dst_entry to ip_forward. Later, after ip_route_input_slow returns, the IP receive handler processing will find this route in the routing cache, de-reference the function pointer, and pass the packet to ip_forward.

```
    rth->u.dst.input = ip_forward;
    rth->u.dst.output = ip_output;
```

Rt_set_nexthop sets information in the route hash entry, including the traffic classifier, tclassid, MTU, metrics, and route type. Next, the routing control flags are set.

```
    rt_set_nexthop(rth, &res, itag);

    rth->rt_flags = flags;
```

If fast routing is configured and the device has the capability, the fast routing control flag is set in the route cache entry.

```
#ifdef CONFIG_NET_FASTROUTE
    if (netdev_fastroute && !(flags&(RTCF_NAT|RTCF_MASQ|RTCF_DOREDIRECT))) {
        struct net_device *odev = rth->u.dst.dev;
        if (odev != dev &&
                dev->accept_fastpath &&
                odev->mtu >= dev->mtu &&
                dev->accept_fastpath(dev, &rth->u.dst) == 0)
            rth->rt_flags |= RTCF_FAST;
}
#endif
```

We are almost done. We call rt_intern_hash to insert the route hash entry into the routing hash table, rt_hash_table.

```
intern:
    err = rt_intern_hash(hash, rth, (struct rtable**)&skb->dst);
```

Now we are done with routing input packets. We decrement the reference count in the input inet_device, in_dev, and if we used it, the output inet_device, out_dev. Fib_res_put actually decrements the reference count in the fib_info structure attached to the fib_res if there is one. When we return a nonzero value, it tells the caller to drop the input packet it is trying to route.

```
done:
    in_dev_put(in_dev);
    if (out_dev)
        in_dev_put(out_dev);
    if (free_res)
        fib_res_put(&res);
out: return err;
```

This label, brd_input, is where we end up if the packet we are trying to route is a broadcast packet or the FIB search returned a broadcast route. If the packet is OK, we set the routing control flags to indicate it is a broadcast, set the specific destination address, and proceed to local input route processing.

```
brd_input:
    if (skb->protocol != __constant_htons(ETH_P_IP))
        goto e_inval;
```

If the source address of the incoming packet is zero, we must set the specific destination address, spec_dst, so a UDP application will know how to set the source address of a response packet.

```
    if (ZERONET(saddr))
        spec_dst = inet_select_addr(dev, 0, RT_SCOPE_LINK);
    else {
```

If the source address wasn't zero, we validate the source address. While we are at it, we also assign the specific destination address, spec_dst.

```
        err = fib_validate_source(saddr, 0, tos, 0, dev, &spec_dst,
                                  &itag);
        if (err < 0)
            goto martian_source;
        if (err)
            flags |= RTCF_DIRECTSRC;
    }
    flags |= RTCF_BROADCAST;
    res.type = RTN_BROADCAST;
    RT_CACHE_STAT_INC(in_brd);
```

The local input label is used for cases where the packet is intended for local consumption. We allocate a route cache entry. This cache entry is also a destination cache entry because a packet using this route will not be sent out any interface. It will be consumed locally.

```
local_input:
    rth = dst_alloc(&ipv4_dst_ops);
    if (!rth)
        goto e_nobufs;
```

We set fields in the routing hash entry. The output function isn't used because we will be consuming these packets, not transmitting them. We set the destination address, ToS, and the source address. The dst_entry flags field is set to DST_HOST.

```
    rth->u.dst.output= ip_rt_bug;
    atomic_set(&rth->u.dst.__refcnt, 1);
    rth->u.dst.flags= DST_HOST;
    if (in_dev->cnf.no_policy)
        rth->u.dst.flags |= DST_NOPOLICY;
    rth->fl.fl4_dst    = daddr;
    rth->rt_dst   = daddr;
    rth->fl.fl4_tos    = tos;
#ifdef CONFIG_IP_ROUTE_FWMARK
    rth->fl.fl4_fwmark = skb->nfmark;
#endif
    rth->fl.fl4_src    = saddr;
    rth->rt_src   = saddr;
#ifdef CONFIG_IP_ROUTE_NAT
    rth->rt_dst_map    = fl.fl4_dst;
    rth->rt_src_map    = fl.fl4_src;
```

```
#endif
#ifdef CONFIG_NET_CLS_ROUTE
    rth->u.dst.tclassid = itag;
#endif
```

The destination device is the loopback device. The input interface is set to the networking interface that received the packet we are trying to route. The most important field of the dst_entry structure is the input function pointer, which is set to ip_local_deliver. This function will receive any input packets using this entry in the routing cache. In addition, if the routing control flags rt_flags has RTCF_LOCAL set, it indicates that the incoming packet is for local consumption.

```
    rth->rt_iif   =
    rth->fl.iif   = dev->ifindex;
    rth->u.dst.dev      = &loopback_dev;
    dev_hold(rth->u.dst.dev);
    rth->rt_gateway     = daddr;
    rth->rt_spec_dst= spec_dst;
    rth->u.dst.input= ip_local_deliver;
    rth->rt_flags = flags|RTCF_LOCAL;
```

The route type in the FIB result is unreachable if we got here from the no_route label, and if so, we indicate an error and unset the local flag.

```
    if (res.type == RTN_UNREACHABLE) {
        rth->u.dst.input= ip_error;
        rth->u.dst.error= -err;
        rth->rt_flags &= ~RTCF_LOCAL;
    }
    rth->rt_type  = res.type;
    goto intern;
```

We came to this label, no_route, if the FIB search indicated that the destination of the packet we were trying to route is unreachable.

```
no_route:
    rt_cache_stat[smp_processor_id()].in_no_route++;
    spec_dst = inet_select_addr(dev, 0, RT_SCOPE_UNIVERSE);
    res.type = RTN_UNREACHABLE;
    goto local_input;
```

If our previous FIB search indicated that the destination address was Martian (entirely unknown), we log the appropriate errors but we don’t cache the route.

```
martian_destination:
    rt_cache_stat[smp_processor_id()].in_martian_dst++;
#ifdef CONFIG_IP_ROUTE_VERBOSE
    if (IN_DEV_LOG_MARTIANS(in_dev) && net_ratelimit())
        printk(KERN_WARNING "martian destination %u.%u.%u.%u from "
                            "%u.%u.%u.%u, dev %s\ n",
                            NIPQUAD(daddr), NIPQUAD(saddr), dev->name);
#endif
e_inval:
```

```
    err = -EINVAL;
    goto done;

e_nobufs:
    err = -ENOBUFS;
    goto done;
```

If the FIB search indicated that the source address was Martian, we also increment the statistics. RFC 1812, Section 5.3.8 has some hints for source address validation based on the reachability of the source addresses through the interface in which the packet came. This is why the code attempts to log the MAC layer header information.

```
martian_source:

    RT_CACHE_STAT_INC(in_martian_src);
#ifdef CONFIG_IP_ROUTE_VERBOSE
    if (IN_DEV_LOG_MARTIANS(in_dev) && net_ratelimit()) {
```

If the source address is Martian, we print out the MAC address. This is because the MAC address may be the only indication that the source address is bogus.

```
        printk(KERN_WARNING "martian source %u.%u.%u.%u from "
                            "%u.%u.%u.%u, on dev %s\ n",
               NIPQUAD(daddr), NIPQUAD(saddr), dev->name);
        if (dev->hard_header_len) {
            int i;
            unsigned char *p = skb->mac.raw;
            printk(KERN_WARNING "ll header: ");
            for (i = 0; i < dev->hard_header_len; i++, p++) {
                printk("%02x", *p);
                if (i < (dev->hard_header_len - 1))
                    printk(":");
            }
            printk("\ n");
        }
    }
#endif
    goto e_inval;
}
```

9.8 Routing Output Packets

In this section, we discuss how IP chooses a route for packets before they are transmitted. If the destination host is directly connected to the sending machine, the destination address can be converted to a link layer address with ARP or another address resolution protocol and the packet can be sent on its way. This seems to be a fairly simple process. However, if the destination is not directly reachable, things are not as simple. The IP protocol must decide how to route the packet, which means that it must choose an output interface if there is more than one network interface device on the sending machine. In addition, it must choose the gateway or next hop that will receive the packet.

Before UDP sends a datagram, it requests a route to the destination address. TCP, however, has already established the route to the destination by the time it sends a packet. TCP requests the route for output packets when the connect socket call is issued by the application. This is because TCP transmission uses the cached route for all segments sent through an open socket once it is active.

Whether TCP or UDP packets are being transmitted, ip_route_connect, defined in file *linux/include/net/route.h*, is the main function for resolving routes for output packets.

```
static inline int ip_route_connect(struct rtable **rp,
                                   u32 daddr, u32 saddr,
                                    u32 tos, int oif, u8 protocol,
                                    u16 sport, u16 dport, struct sock *sk)
{
```

As with the input route resolving functions, the flowi structure, examined earlier, is initialized with the information to try to do a match of routes in the route cache. It is also used for a search of the FIB if the route can't be matched in the cache.

```
    struct flowi fl = {  .oif = oif,
                         .nl_u = {  .ip4_u = {  .daddr = dst,
                                                .saddr = src,
                                                .tos   = tos }  } ,
                             .proto = protocol,
                         .uli_u = {  .ports =
                                   {  .sport = sport,
                                     .dport = dport }  }  } ;
    int err;
    if (!dst || !src) {
```

We call one of the fast path routing functions. Afterwards, we try to update both the source and destination addresses if one is missing.

```
        err = __ip_route_output_key(rp, &fl);
        if (err)
            return err;
        fl.fl4_dst = (*rp)->rt_dst;
        fl.fl4_src = (*rp)->rt_src;
        ip_rt_put(*rp);
        *rp = NULL;
    }
```

If both the source and destination addresses are defined, we pass in a pointer to the sock in sk.

```
    return ip_route_output_flow(rp, &fl, sk, 0);
}
```

The function ip_route_output_flow can automatically transform the route if the protocol value in the flowi structure is set to NAT or some other nonzero number.

```
int ip_route_output_flow(struct rtable **rp, struct flowi *flp, struct sock
                         *sk, int flags)
```

```
{
    int err;
```

We call the fast route resolving function here.

```
    if ((err = __ip_route_output_key(rp, flp)) != 0)
        return err;
```

If we found a route, and the proto info in the flowi was not zero, we try to transform the route.

```
    return flp->proto ? xfrm_lookup((struct dst_entry**)rp, flp, sk,
flags) : 0;
}
```

_ip_route_output_key is the function that does the fast path output routing. First, it tries to find a matching route in the route cache. If it can't find the route in the cache, it tries to find the route by searching the FIB. This function returns a zero if the route is found, and a nonzero value if it is not. If the search was successful, a pointer to the route cache entry, rtable, in placed in the parameter, rp.

```
int ip_route_output_key(struct rtable **rp, const struct flowi *flp)
{
```

We use a simple 32-bit hash for a first-level search of the hash table. Once a slot in the hash table, rt_hash_table, is identified, we read-lock the table location and try for an exact match of the routes at that location.

```
    unsigned hash;
    struct rtable *rth;
    hash = rt_hash_code(flp->fl4_dst, flp->fl4_src ^ (flp->oif << 5),
                        flp->fl4_tos);
    rcu_read_lock();
```

This is a second-level search done with the information from the flowi structure pointed to by the argument, flp. In this search, we try to find an exact match for the route. In many cases, there will be only one rtable entry at a hash slot.

```
    for (rth = rt_hash_table[hash].chain; rth; rth = rth->u.rt_next) {
        smp_read_barrier_depends();
        if (rth->fl.fl4_dst == flp->fl4_dst &&
            rth->fl.fl4_src == flp->fl4_src &&
            rth->fl.iif == 0 &&
            rth->fl.oif == flp->oif &&
#ifdef CONFIG_IP_ROUTE_FWMARK
            rth->fl.fl4_fwmark == flp->fl4_fwmark &&
#endif
```

Let's look at this part of the if statement closely. The bit definitions of the fl4_tos field in the flowi structure are similar to the ToS field of the IP header. Earlier in this chapter, we examined the flowi structure. In the fl4_tos field, the RTO_ONLINK actually refers to bit zero, which is not defined in the "real" IP packet ToS field. Linux uses it here to identify a route to a directly

connected host (reachable via link layer transmission). Generally, when we are called from ARP, tos is set to RTO_ONLINK to request a "route" to a directly connected host. Because the routing table is derived from the destination cache, the search of the routing cache returns a destination cache entry pointing to the attached host.

```
            !((rth->fl.fl4_tos ^ flp->fl4_tos) &
               (IPTOS_RT_MASK | RTO_ONLINK))) {
```

If the key matches, we set lastuse to the current time to indicate when this destination entry was last used. In addition, the use count is decremented. We return a pointer to the new destination cache entry in rp and exit from the function.

```
            rth->u.dst.lastuse = jiffies;
            dst_hold(&rth->u.dst);
            rth->u.dst.__use++;
            RT_CACHE_STAT_INC(out_hit);
            rcu_read_unlock();
            *rp = rth;
            return 0;
        }
        RT_CACHE_STAT_INC(out_hlist_search);
    }
```

If we didn’t find a match in the route cache, we unlock the hash table slot and call the slow output route resolving function, ip_route_output_slow, with flp to search the FIB.

```
    rcu_read_unlock();

    return ip_route_output_slow(rp, flp);
}
```

9.8.1 The Main Output Route Resolving Function

If the fast path route failed, we continue with the slow path routing. If the fast path routing function can’t match the new route in the route cache, it calls the function ip_route_output_slow to search the FIB.

```
int ip_route_output_slow(struct rtable **rp, const struct flowi *oldflp)
{
```

This variable, tos, is built from the tos field in the flowi structure pointed to by the parameter oldflp. It mostly contains the IP ToS field bits, which are used as part of the criteria to determine the route. However, tos also includes another bit that is not part if the IP header ToS field. When set in fl4_tos field of flowi, this bit, RTO_ONLINK, defined as bit zero, indicates that the route is to a directly connected host.

```
    u32 tos = oldflp->fl4_tos & (IPTOS_RT_MASK | RTO_ONLINK);
```

In this function, we create a new flowi structure, which is created from the input flowi pointed to by flp. Most of the fields are copied from the old flp but a few are calculated.

```
    struct flowi fl = {  .nl_u = {  .ip4_u =
                                 {  .daddr = oldflp->fl4_dst,
                                    .saddr = oldflp->fl4_src,
```

In the tos field in the IPv4 part of the flowi structure, the RTO_ONLINK actually refers to bit zero, which is not defined in the "real" IP packet ToS field. Linux uses it here to identify a route to a directly connected host (reachable via link layer transmission).

```
                                    .tos = tos & IPTOS_RT_MASK,
                                    .scope = ((tos & RTO_ONLINK) ?
                                              RT_SCOPE_LINK :
                                              RT_SCOPE_UNIVERSE),
```

The next field, fwmark, is for firewall marks. It is used for traffic shaping if firewall marks are configured into the Linux kernel.

```
#ifdef CONFIG_IP_ROUTE_FWMARK
                                    .fwmark = oldflp->fl4_fwmark
#endif
                                  }  } ,
                           .iif = loopback_dev.ifindex,

                           .oif = oldflp->oif } ;
```

The structure fib_result holds the result of a search of the FIB.

```
    struct fib_result res;
    unsigned flags = 0;
    struct rtable *rth;
```

We will use dev_out as a pointer to the input network interface device associated with the route, and in_dev as a pointer to the output network interface.

```
    struct net_device *dev_out = NULL;
    struct in_device *in_dev = NULL;
    unsigned hash;
    int free_res = 0;
    int err;
    res.fi        = NULL;
#ifdef CONFIG_IP_MULTIPLE_TABLES
    res.r         = NULL;
#endif
```

We do some basic checking before looking up the route.

```
    if (oldflp->fl4_src) {
```

If the source address is specified by the caller, we check to see if it is a Martian address.

```
        err = -EINVAL;
        if (MULTICAST(oldflp->fl4_src) ||
            BADCLASS(oldflp->fl4_src) ||
            ZERONET(oldflp->fl4_src))
```

```
            goto out;
```

In this test, we check to see if the source address is one of our local addresses. If the address is assigned to an interface, ip_dev_find returns the network interface device that has that address. This check is functionally similar to calling inet_addr_type with source address as an argument to see if the address is in the local FIB table.

```
        dev_out = ip_dev_find(oldflp->fl4_src);
        if (dev_out == NULL)
            goto out;
```

Comments in the code say that code was removed to see if the output interface oif was for the same device as the one returned by ip_dev_find. This check would have been incorrect because the source address could have been an address different from the device from which the packets are sent.

```
        if (oldflp->oif == 0
            && (MULTICAST(oldflp->flr_dst) ||
            oldflp->fl4_dst == 0xFFFFFFFF)) {
```

Comments in the code say this is a special hack, so applications can send packets to multicast and broadcast addresses without specifying the IP_PKTINFO IP option. This facilitates applications such as VIC and VAT for video and audio conferencing. We need this hack because these applications bind the socket to the loopback address, set the multicast TTL to zero, and send the packets out without doing a join group or specifying the outgoing multicast interface.

```
            fl.oif = dev_out->ifindex;
```

Now we are ready to go to the label make_route to enter the new route in cache.

```
            goto make_route;
        }
```

This is just a cleanup. We decrement the use count of the device we used temporarily, dev_out.

```
        if (dev_out)
            dev_put(dev_out);
        dev_out = NULL;
    }
```

If the output interface, oif, is specified by the caller, we get the net_device pointer and make sure that it references an in_device structure that contains the AF_INET type address information. The in_device structure is explained in Section 6.9 of Chapter 6.

```
    if (oldflp->oif) {
        dev_out = dev_get_by_index(oldflp->oif);
        err = -ENODEV;
        if (dev_out == NULL)
            goto out;
        if (__in_dev_get(dev_out) == NULL) {
            dev_put(dev_out);
```

```
            goto out;
        }
```

Now we check to see if we are trying to route a multicast destination address, and if so, we get the source address by calling inet_select_addr. This source address will be put in the route information so it can be used later as the source address of outgoing packets sent to the multicast or broadcast address in flp->fl4_dst. Next, since there is no need to do a FIB lookup, we go to make_route to enter the new route in the route cache.

```
        if (LOCAL_MCAST(oldflp->fl4_dst) || oldflp->fl4_dst ==
0xFFFFFFFF) {
            if (!fl.fl4_src)
                fl.fl4__src = inet_select_addr(dev_out, 0,
                                               RT_SCOPE_LINK);
            goto make_route;
        }
        if (!fl.fl4_src) {
            if (MULTICAST(oldflp->fl4_dst))
                fl.fl4_src = inet_select_addr(dev_out, 0, fl.fl4_scope);
            else if (!oldflp->fl4_dst)
                fl.fl4_src = inet_select_addr(dev_out, 0,
                                              RT_SCOPE_HOST);
        }
    }
```

If the destination address in the key is NULL, we assume that we are looking for a local (internal) destination. We set the "output" network device to the loopback device and set both flags to look for a local route. Therefore, packets using this route will be looped back; they will be sent back up the IP stack. There is no need to do the FIB lookup for this key, so we go straight to make_route, which enters this local route in the route cache.

```
    if (!fl.fl4_dst) {
        fl.fl4_dst = fl.fl4_src;
        if (!fl.fl4_dst)
            fl.fl4_dst = fl.fl4_src = htonl(INADDR_LOOPBACK);
        if (dev_out)
            dev_put(dev_out);
        dev_out = &loopback_dev;
        dev_hold(dev_out);
        fl.oif = loopback_dev.ifindex;
        res.type = RTN_LOCAL;
        flags |= RTCF_LOCAL;
        goto make_route;
    }
```

Now, we call fib_lookup to get the route for fl. If it finds a route, it returns a zero and puts the result in res.

```
    if (fib_lookup(&fl, &res)) {
```

The FIB lookup has failed to find a route.

```
        res.fi = NULL;
```

```
        if (oldflp->oif) {
```

We are here because the FIB lookup failed even though an output interface was specified in the lookup key, oldkey. A comment in the code states that the routing tables must be wrong if the lookup failed event even though an output device was specified. We are allowed to send packets out an interface even when there are no routes specifying the interface and no addresses assigned to the interface. Therefore, we assume that the destination is directly connected to the output interface oif even though there was no route in the FIB. If the output interface, oif is specified, the route lookup is only for checking to see whether the final destination is directly connected or reachable only through a gateway.

```
            if (fl.fl4_src == 0)
```

If the source address wasn't specified, we must pick one. We put it in the key, set the route type to multicast, and jump to make_route to enter the route into cache.

```
                fl.fl4_src = inet_select_addr(dev_out, 0,
                                              RT_SCOPE_LINK);
            res.type = RTN_UNICAST;
            goto make_route;
         }
```

We arrived here because we couldn't find a route, so we set the error to unreachable and get out.

```
        if (dev_out)
            dev_put(dev_out);
        err = -ENETUNREACH;
        goto out;
    }
```

This section of the code is executed if we know that the FIB lookup has given us a route. We do some checks of the route type in the type field before we add the route to the cache.

```
    free_res = 1;
```

This is a check if the route we got from the FIB indicated NAT. This shouldn't happen, so we get out.

```
    if (res.type == RTN_NAT)
        goto e_inval;
```

We got here because the FIB lookup gave us a "local" route, which means that the destination address in the key was one of our own addresses. We set the output "device" to the loopback device, and the route cache flag, flags, to RTCF_LOCAL indicating that the route is a local route. Next, we go to make_route to enter the route in cache.

```
    if (res.type == RTN_LOCAL) {
        if (!fl.fl4_src)
            fl.fl4_src = fl.fl4_dst;
        if (dev_out)
            dev_put(dev_out);
```

```
        dev_out = &loopback_dev;
        dev_hold(dev_out);
        fl.oif = dev_out->ifindex;
        if (res.fi)
            fib_info_put(res.fi);
        res.fi = NULL;
        flags |= RTCF_LOCAL;
        goto make_route;
    }
```

Multipath routing is a kernel option that allows more than one routing path to be defined to the same destination. If the option is configured, we check the fib_nhs field in the fib_info structure to see if it is greater than one. This lets us know that there is more than one "next hop" for the same destination.

```
#ifdef CONFIG_IP_ROUTE_MULTIPATH
    if (res.fi->fib_nhs > 1 && key.oif == 0)
        fib_select_multipath(&key, &res);
    else
#endif
```

Here, we check to see if we got a fib_result with no netmask and no output device. With neither of these things, we get the default route by calling fib_select_default. The field prefixlen specifies the number of bits to use for the netmask, and if it is zero, there is no netmask. If the output device was specified in the fib_result we would know we are trying to reach a directly connected host, and if the netmask was specified, we would know we have a route to a gateway.

```
if (!res.prefixlen && res.type == RTN_UNICAST && !fl.oif)
        fib_select_default(&fl, &res);

    if (!fl.fl4_src)
        fl.fl4_src = FIB_RES_PREFSRC(res);

    if (dev_out)
        dev_put(dev_out);
    dev_out = FIB_RES_DEV(res);
    dev_hold(dev_out);
    fl.oif = dev_out->ifindex;
```

At this point, we are done with the FIB lookup. We have a route so we enter it in the route cache after making a few checks for broadcast, multicast, and just plain bad destination addresses. It may seem that some of these checks are redundant at this point, but we may have come to this label by bypassing the FIB lookup.

```
make_route:
    if (LOOPBACK(fl.fl4_src) && !(dev_out->flags&IFF_LOOPBACK))
```

If the source address is the loopback address, the output device must also be a loopback device.

```
        goto e_inval;
```

Here we set the route type to RTN_BROADCAST or RTN_MULTICAST depending on the destination address.

```
if (key.dst == 0xFFFFFFFF)
    res.type = RTN_BROADCAST;
else if (MULTICAST(fl.fl4_dst))
    res.type = RTN_MULTICAST;
```

If the destination address is Martian, we get out.

```
else if (BADCLASS(fl.fl4_dst) || ZERONET(fl.fl4_dst))
    goto e_inval;
```

If the output network interface is the loopback device, we must have a local route, so we set the route control flags to RTCF_LOCAL.

```
if (dev_out->flags & IFF_LOOPBACK)
    flags |= RTCF_LOCAL;
```

Next, we get the in_device structure for the output network interface because it contains the list of multicast addresses for the output device.

```
in_dev = in_dev_get(dev_out);
if (!in_dev)
    goto e_inval;
```

We don’t need fib_info for either broadcast or local routes, so fi is set to NULL and fib_info is freed.

```
if (res.type == RTN_BROADCAST) {
    flags |= RTCF_BROADCAST | RTCF_LOCAL;
    if (res.fi) {
        fib_info_put(res.fi);
        res.fi = NULL;
    }
```

There are a few additional things we must do for multicast routes.

```
}  else if (res.type == RTN_MULTICAST) {
```

First, we initialize the route cache flags to indicate both multicast and local because we have a multicast route. Before entering the route in cache, we decide whether we will loop back packets sent via this multicast route. We do this by checking to see if the destination address is in the list of multicast groups for our output interface. If not, we reset the RTCF_LOCAL so multicast packets sent via this route won’t be looped back.

```
        flags |= RTCF_MULTICAST|RTCF_LOCAL;
        read_lock(&inetdev_lock);
        if (ip_check_mc(in_dev, oldflp->fl4_dst,oldflp->fl4_src,
             oldflp->proto))
            flags &= ~RTCF_LOCAL;
```

Here, a comment in the code says that this is a hack. If the multicast route does not exist, use the default multicast route but do not send the packet to a gateway.

```
        if (res.fi && res.prefixlen < 4) {
            fib_info_put(res.fi);
            res.fi = NULL;
        }
    }
```

We call dst_alloc to allocate a route cache entry from the generic destination cache. We increment the reference count in the cache entry. Next, we set fields in the new route cache entry including the flowi values.

```
    rth = dst_alloc(&ipv4_dst_ops);
    if (!rth)
        goto e_nobufs;
    atomic_set(&rth->u.dst.__refcnt, 1);
    rth->u.dst.flags= DST_HOST;
    if (in_dev->cnf.no_xfrm)
        rth->u.dst.flags |= DST_NOXFRM;
    if (in_dev->cnf.no_policy)
        rth->u.dst.flags |= DST_NOPOLICY;
    rth->fl.fl4_dst     = oldflp->fl4_dst;
    rth->fl.fl4_tos     = tos;
    rth->fl.fl4_src     = oldflp->fl4_src;
    rth->fl.oif   = oldflp->oif;
#ifdef CONFIG_IP_ROUTE_FWMARK
    rth->fl.fl4_fwmark= oldflp->fl4_fwmark;
#endif
    rth->rt_dst  = fl.fl4_dst;
    rth->rt_src  = fl.fl4_src;
#ifdef CONFIG_IP_ROUTE_NAT
    rth->rt_dst_map     = fl.fl4_dst;
    rth->rt_src_map     = fl.fl4_src;
#endif
```

We set the input interface for the route. Next, we set the output device specified in the destination cache entry. Later, when packets are transmitted via this route, the output interface device in dev_out can be accessed quickly via the destination cache entry through the dst field of the socket buffer.

```
    rth->rt_iif = oldflp->oif ? : dev_out->ifindex;
    rth->u.dst.dev      = dev_out;
    dev_hold(dev_out);
    rth->rt_gateway = fl.fl4_dst;
    rth->rt_spec_dst= fl.fl4_src;
```

Since this is an output route, ip_output is the output function for this route. When a transmit protocol sends the packet, it will call ip_output through the output field of the destination cache entry. We increment the statistics for the number of slow routes.

```
    rth->u.dst.output=ip_output;
    RT_CACHE_STAT_INC(out_slow_tot);
```

If the route cache flag indicates that this is a local route, the input function is set. Later, when the IP receive function gets a packet via this route, it will call ip_local_deliver through the destination cache entry's input field.

```
    if (flags & RTCF_LOCAL) {
        rth->u.dst.input = ip_local_deliver;
        rth->rt_spec_dst = fl.fl4_dst;
    }
```

If the flags indicate that this is a broadcast or multicast route, the output function is set to ip_mc_output. This function is discussed in Section 9.13.

```
    if (flags & (RTCF_BROADCAST | RTCF_MULTICAST)) {
        rth->rt_spec_dst = fl.fl4_src;
```

If the route is to a multicast address, the output function pointer in the destination cache is set to point to the multicast output routing function.

```
        if (flags & RTCF_LOCAL && !(dev_out->flags & IFF_LOOPBACK)) {
            rth->u.dst.output = ip_mc_output;
            RT_CACHE_STAT_INC(out_slow_mc);
        }
#ifdef CONFIG_IP_MROUTE
        if (res.type == RTN_MULTICAST) {
            if (IN_DEV_MFORWARD(in_dev) &&
                !LOCAL_MCAST(oldflp->fl4_dst)) {
                rth->u.dst.input = ip_mr_input;
                rth->u.dst.output = ip_mc_output;
            }
        }
#endif
    }
```

We call rt_set_nexthop to set some information about the gateway from the route cache. We get this from the fib_info structure attached to res. Then, we set the routing cache rt_flags field.

```
    rt_set_nexthop(rth, &res, 0);
    rth->rt_flags = flags;
```

Next, we calculate the hash code for the new route cache entry and enter rth into the route cache by calling rt_intern_hash.

```
    hash = rt_hash_code(oldflp->fl4_dst, oldflp->fl4_src ^
                        (oldflp->oif << 5), tos);
    err = rt_intern_hash(hash, rth, rp);
done:
    if (free_res)
        fib_res_put(&res);
    if (dev_out)
        dev_put(dev_out);
out: return err;

e_inval:
```

```
    err = -EINVAL;
    goto done;
e_nobufs:
    err = -ENOBUFS;
    goto done;
}
```

9.9 Internet Peers and The Inet_Peer Structure

The Internet peer structure is used to generate the value in the identification field of outgoing packets. IP fragments outgoing packets when the packet size exceeds the MTU of the route, so it must provide a way for the receiving machine to identify the packet fragments. The receiving uses the identification field in the IP header of the received fragment when reassembling the original packet. Since IP datagrams are not guaranteed to arrive in order, an unique sequential ID is generated for each outgoing fragment so the machine receiving the packets can piece together the original packet from the fragments correctly. Each outgoing packet sent to a specific destination has an ID that is incremented by one from the previously transmitted packet sent to the same destination. Linux IP uses the Internet peer to keep track of each of the known peers so the IDs can be quickly calculated from the route table entry without having to do a separate search of the route cache or the FIB.

The Internet peers are kept on an AVL tree, and each node is an instance of the inet_peer structure. This structure is defined in file *linux/include/net/inetpeer.h*.

```
struct inet_peer
{
```

Avl_left and avl_right position the node in the AVL tree. Unused_next andunused_prevp are to maintain the node in the list of unused nodes.

```
    struct inet_peer    *avl_left, *avl_right;
    struct inet_peer    *unused_next, **unused_prevp;
```

Refcnt is the reference count for this node. It must be atomically incremented and decremented. The next field, dtime, is the time when this entry was last referenced.

```
    atomic_t            refcnt;
    unsigned long       dtime;
```

V4daddr is the peer's IPv4 address.

```
    __u32               v4daddr;
    __u16               avl_height;
```

The next field, ip_id_count, is the value for the identification field in the next IP packet.

```
    __u16               ip_id_count;
```

The next two fields, tcp_ts and tcp_ts_stamp, are used for TCP **TIME_WAIT** state recycling, where sockets in the **TIME_WAIT** state can be reused as new connections are requested. Old timestamp values are saved in here for at least the 2MSL interval for the **TIME_WAIT** state duration. When a new connection is requested, the timestamp values are restored from these values. See Chapter 10 for more information about the **TIME_WAIT** state in TCP.

```
    __u32               tcp_ts;
    unsigned long       tcp_ts_stamp;
} ;
```

Each inet_peer instance can be reached through the peer field of the rtable structure. There is exactly one inet_peer instance for each peer IP address. The inet_peer entry is accessed only when the IP needs to select a value for the identification field of an outgoing IP packet. Normally, inet_peer nodes are only removed when the reference count, refcnt, becomes zero. The node isn't removed immediately, but only after a certain amount of time has passed since it was last referenced. When a new route cache entry is created, it can find the inet_peer by searching the AVL tree. Least recently used entries may be cleaned off the tree if we are out of space or the pool of entries is overloaded. The advantage of using an AVL tree over a hash table is to reduce the risk of successful Denial-of-Service (DoS) attacks. DoS attacks can overload a single hash bucket, causing time to be wasted with linear searches. Unfortunately, since the route cache implementation uses a hash table and there is mostly a one-to-one relationship between route table entries and the inet_peer entries, the theoretical advantages of having the inet_peers in an AVL tree may not be fully realized.

9.10 The Address Resolution Protocol

The ARP protocol maps IP addresses to link layer addresses [RFC 826]. It is defined for the Ethernet Local Area Network (LAN). When a packet is about to be transmitted, the output routine prepends the link layer header to the output packet. The host contains a lookup table called the *ARP cache* that contains the mapping of the destination IP address and the link layer destination hardware address. When a lookup in the ARP cache fails, an ARP request message is broadcast on the local area net. When an ARP response message is received, the sender will update its ARP cache.

Any implementation of TCP/IP needs both an ARP cache and a routing table. Some systems implement a separate unique table for the ARP cache that doesn't share any of the structure with the routing table. In Linux, ARP is implemented in file *linux/net/ipv4/arp.c*. It uses the generic neighbor cache facility, described in Chapter 6, as the framework for the ARP cache. Some of this framework is shared with the routing cache implementation. Most of the ARP table management is supplied by the neighbor facility, and most of the generic framework for the neighbor cache was discussed in Chapter 6. However, in this section, we will show the parts that are specific to the ARP protocol. We show how the ARP-specific parts are initialized, how ARP processes incoming packets, and how it handles requests to resolve addresses.

9.10.1 ARP Protocol Initialization

The ARP protocol has a few structures that are initialized at compile time in the file *linux/net/ipv4/arp*. The first of these is the packet_type structure, which registers the handler

function, arp_rcv, which will be called when an ARP packet arrives at any network interface device. Chapter 6 has more details about how packet handlers are registered.

```
static struct packet_type arp_packet_type = {
    .type = __constant_htons(ETH_P_ARP),
    .func = arp_rcv,
} ;
```

Another structure is the net device notifier block for the ARP protocol, arp_netdev_notifier.

```
struct notifier_block arp_netdev_notifier = {
    .notifier_call = arp_netdev_event,
} ;
```

The function arp_netdev_event will be called with any change in device status.

The ARP protocol is initialized by the function arp_init implemented in file *linux/net/ipv4/arp.c*. This function is called as part of the main IP initialization process.

```
void __init arp_init(void)
{
```

The ARP cache is initialized by calling neigh_table_init. ARP uses the neighbor facility, and neigh_table_init is the neighbor cache initialization function. Here we are creating a neighbor table instance. The neig_table instance for ARP, arp_tbl, is initialized at compile time and is shown later in this section.

```
    neigh_table_init(&arp_tbl);
```

The ARP protocol is not encapsulated with an IP packet. It is encapsulated directly by the link layer header; therefore, we must call dev_add_pack to register our packet type with the queuing layer. As shown earlier, the handler function points to arp_rcv, which will be called when a packet of type ETH_P_ARP is received at the network interface

```
    dev_add_pack(&arp_packet_type);
```

We call arp_proc_init to initialize the ARP "directory" in the /proc pseudo file-system. This is so users can look at the ARP cache and tuning parameters. Chapter 6 contains more information about the /proc filesystem.

```
    arp_proc_init();
```

There are many ARP parameters that can be tuned. We call the following function so that the ARP parameters can be set via sysctl if it is configured into the kernel.

```
#ifdef CONFIG_SYSCTL
    neigh_sysctl_register(NULL, &arp_tbl.parms, NET_IPV4,
                          NET_IPV4_NEIGH, "ipv4");
#endif
```

Next, we register with the netdevice notifier so ARP will be informed when the interface address changes. When an interface address changes, the function arp_netdev_event will be called. We will see later that we are only interested in the NETDEV_CHANGEADDR notification. We don't use the notification of an interface going down. Instead, the function arp_ifdown is called directly to tell about this event.

```
    register_netdevice_notifier(&arp_netdev_notifier);
}
```

Now let's look at how the ARP cache is instantiated. As discussed earlier, Linux uses the neighbor cache as the framework for the ARP cache. See Chapter 6 for more details about the neighbor system and the neigh_table structure. Next, we show how a neigh_table structure instance, arp_table is initialized at compile time. This structure is the actual ARP cache.

```
struct neigh_table arp_tbl = {
    .family =     AF_INET,
    .entry_size = sizeof(struct neighbour) + 4,
    .key_len =    4,
```

The neigh_table has a destructor function defined but it is used for ARP. This is because the ARP protocol is a permanent part of IPv4, therefore the ARP cache will never totally be removed.

```
    .hash =       arp_hash,
    .constructor =arp_constructor,
    .proxy_redo = parp_redo,
    .id =         "arp_cache",
```

Here is the initialization of the neigh_parms values that contain the timeout values and the garbage collection intervals.

```
    .parms = {
        .tbl =                      &arp_tbl,
        .base_reachable_time =      30 * HZ,
        .retrans_time = 1 * HZ,
        .gc_staletime = 60 * HZ,
        .reachable_time =           30 * HZ,
        .delay_probe_time =   5 * HZ,
        .queue_len =          3,
        .ucast_probes = 3,
        .mcast_probes =       3,
        .anycast_delay =      1 * HZ,
        .proxy_delay =        (8 * HZ) / 10,
        .proxy_qlen =         64,
        .locktime =           1 * HZ,
    } ,
    .gc_interval =      30 * HZ,
    .gc_thresh1 =       128,
    .gc_thresh2 =       512,
    .gc_thresh3 =       1024,
} ;
```

There are also four neighbor operations structures initialized for ARP, arp_generic_ops, arp_hh_ops, arp_direct_ops, and arp_broken_arps. Arp_generic_ops contain the operation

functions called when we actually must resolve an address. Arp_generic_ops is initialized as follows.

```
static struct neigh_ops arp_generic_ops = {
    .family =           AF_INET,
```

Only the solicit and error_report fields are set to functions unique to ARP. The other fields use the default ones for the neighbor cache.

```
    .solicit =          arp_solicit,
    .error_report =     arp_error_report,
    .output =           neigh_resolve_output,
    .connected_output = neigh_connected_output,
    .hh_output =        dev_queue_xmit,
    .queue_xmit =       dev_queue_xmit,
} ;
```

The arp_hh_ops structure is initialized as shown here. Generally, this set of operations is used with Ethernet.

```
static struct neigh_ops arp_hh_ops = {
    .family =           AF_INET,
```

Again, the solicit and error_report functions are unique to ARP. The dev_queue_xmit function will queue the packet for transmission out of a network interface device. It will be called once a neighbor entry is resolved to an actual Ethernet address. The other operations are set to the generic ones for the neighbor cache.

```
    .solicit =          arp_solicit,
    .error_report =     arp_error_report,
    .output =           neigh_resolve_output,
    .connected_output = neigh_resolve_output,
    .hh_output =        dev_queue_xmit,
    .queue_xmit =       dev_queue_xmit,
} ;
```

The arp_direct_ops are used for a directly connected host that requires no hardware address, such as a point-to-point link. This is why all the functions point to dev_queue_xmit.

```
static struct neigh_ops arp_direct_ops = {
    .family =           AF_INET,
    .output =           dev_queue_xmit,
    .connected_output = dev_queue_xmit,
    .hh_output =        dev_queue_xmit,
    .queue_xmit =       dev_queue_xmit,
} ;
```

Arp_broken_ops is used for unreachable destinations.

```
struct neigh_ops arp_broken_ops = {
    .family =           AF_INET,
    .solicit =          arp_solicit,
    .error_report =     arp_error_report,
```

```
    .output =           neigh_compat_output,
    .connected_output = neigh_compat_output,
    .hh_output =        dev_queue_xmit,
    .queue_xmit =       dev_queue_xmit,
} ;
```

9.10.2 Receiving and Processing ARP Packets

The function arp_rcv implemented in the file *linux/net/ipv4/arp.c* is called when an ARP packet arrives at an Ethernet or other interface that supports ARP.

```
int arp_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type
*pt)
    struct arphdr *arp;
```

First, we make sure that the incoming packet is at least long enough to hold the ARP header. If not, we free the packet and get out.

```
    if (!pskb_may_pull(skb, (sizeof(struct arphdr) +
                       (2 * dev->addr_len) +
                       (2 * sizeof(u32)))))
        goto freeskb;
```

In this function, we check to make sure that the interface through which the packet arrived uses the ARP protocol and that the packet is not a loopback packet.

```
    if (arp->ar_hln != dev->addr_len ||
        dev->flags & IFF_NOARP ||
        skb->pkt_type == PACKET_OTHERHOST ||
        skb->pkt_type == PACKET_LOOPBACK ||
        arp->ar_pln != 4)
        goto freeskb;
```

Next, we check to make sure that the incoming packet is not shared. This would be the case if another protocol had received the packet and was modifying it. This is unlikely, but possible.

```
    if ((skb = skb_share_check(skb, GFP_ATOMIC)) == NULL)
        goto out_of_mem;
```

If all is OK, we call arp_process through the net filter.

```
    return NF_HOOK(NF_ARP, NF_ARP_IN, skb, dev, NULL, arp_process);
freeskb:
    kfree_skb(skb);
out_of_mem:
    return 0;
}
```

The next function, arp_process, also implemented in the file *linux/net/ipv4/arp.c*, handles all arp_request packets. It is called from the low-level packet handler function arp_rcv.

```
int arp_process(struct sk_buff *skb)
```

```
    struct net_device *dev = skb->dev;
    struct in_device *in_dev = in_dev_get(dev);
```

Arp points to the ARP header in the incoming packet.

```
    struct arphdr *arp;
    unsigned char *arp_ptr;
    struct rtable *rt;
```

Sha points to the source hardware address, and tha points to the target hardware address.

```
    unsigned char *sha, *tha;
```

Sip and Tip are for the source IP and destination IP addresses specified in the ARP packet.

```
    u32 sip, tip;
    u16 dev_type = dev->type;
    int addr_type;
    struct neighbour *n;
```

First, we verify the packet header and make sure that the incoming network interface is an ARP-type interface. We allow for ARP packets from Ethernet, Token Ring, and FDDI.

```
    if (in_dev == NULL)
        goto out;

    arp = skb->nh.arph;
    switch (dev_type) {
    default:
        if (arp->ar_pro != htons(ETH_P_IP) ||
            htons(dev_type) != arp->ar_hrd)
            goto out;
        break;
```

This switch contains cases for each of the devices that are defined to support ARP.

```
#ifdef CONFIG_NET_ETHERNET
    case ARPHRD_ETHER:
#endif
#ifdef CONFIG_TR
    case ARPHRD_IEEE802_TR:
#endif
#ifdef CONFIG_FDDI
    case ARPHRD_FDDI:
#endif
```

In addition, we check for IEEE 802-type headers.

```
#ifdef CONFIG_NET_FC
    case ARPHRD_IEEE802:
#endif
#if defined(CONFIG_NET_ETHERNET) || defined(CONFIG_TR) || \
    defined(CONFIG_FDDI)         || defined(CONFIG_NET_FC)
```

If the framing is IEEE 802 type framing, we accept hardware types of both ARPHRD_ETHER, one, and ARPHRD_IEEE802, six. This is also true for Fibre Channel (FDDI) [RFC 2625]. The ARP hardware types are defined in file *linux/include/linux/if_arp.h*.

```
        if ((arp->ar_hrd != htons(ARPHRD_ETHER) &&
             arp->ar_hrd != htons(ARPHRD_IEEE802)) ||
             arp->ar_pro != htons(ETH_P_IP))
            goto out;
        break;
#endif
#if defined(CONFIG_AX25) || defined(CONFIG_AX25_MODULE)
    case ARPHRD_AX25:
        if (arp->ar_pro != htons(AX25_P_IP) ||
            arp->ar_hrd != htons(ARPHRD_AX25))
            goto out;
        break;
#if defined(CONFIG_NETROM) || defined(CONFIG_NETROM_MODULE)
    case ARPHRD_NETROM:
        if (arp->ar_pro != htons(AX25_P_IP) ||
            arp->ar_hrd != htons(ARPHRD_NETROM))
            goto out;
        break;
#endif
#endif
    }
```

We only accept ARP request and ARP reply type messages. This is OK because there really aren't any other ARP message types.

```
    if (arp->ar_op != htons(ARPOP_REPLY) &&
        arp->ar_op != htons(ARPOP_REQUEST))
        goto out;
```

Next, we extract some fields from the ARP header.

```
    arp_ptr= (unsigned char *)(arp+1);
    sha     = arp_ptr;
    arp_ptr += dev->addr_len;
    memcpy(&sip, arp_ptr, 4);
    arp_ptr += 4;
    tha     = arp_ptr;
    arp_ptr += dev->addr_len;
    memcpy(&tip, arp_ptr, 4);
```

Now we check for strange ARP requests such as requests for the loopback address or multicast addresses.

```
    if (LOOPBACK(tip) || MULTICAST(tip))
        goto out;
```

This is a special case where we must set the Frame Relay (FR) source (Q.922) address.

```
    if (dev_type == ARPHRD_DLCI)
        sha = dev->broadcast;
```

At this point, we handle ARP request packets first. First, we handle the special case for duplicate address detection. We check to see if the source IP address in the request packet is zero and the target IP address is one of our local addresses. If so, we can send the response right away.

```
    if (sip == 0) {
        if (arp->ar_op == htons(ARPOP_REQUEST) &&
            inet_addr_type(tip) == RTN_LOCAL)
            arp_send(ARPOP_REPLY,ETH_P_ARP,tip,dev,tip,sha,dev->
dev_addr,dev->dev_addr);
        goto out;
    }
```

Now we handle the general case for the ARP request message type. We validate the address by checking to make sure that we have a route to the address being requested.

```
    if (arp->ar_op == htons(ARPOP_REQUEST) &&
        ip_route_input(skb, tip, sip, 0, dev) == 0) {
        rt = (struct rtable*)skb->dst;
        addr_type = rt->rt_type;
```

Neigh_event_ns does a lookup in the ARP cache for the source hardware address. If found, it updates the neighbor cache entry.

```
        if (addr_type == RTN_LOCAL) {
            n = neigh_event_ns(&arp_tbl, sha, &sip, dev);
            if (n) {
                int dont_send = 0;
                if (IN_DEV_ARPFILTER(in_dev))
                    dont_send |= arp_filter(sip,tip,dev);
                if (!dont_send)
```

Here is where we send out the ARP reply message if, of course, we found the entry in the ARP cache.

```
                    arp_send(ARPOP_REPLY,ETH_P_ARP,sip,dev,tip,sha,
                             dev->dev_addr,sha);
                neigh_release(n);
            }
            goto out;
```

This is a check to see if we are receiving a proxy ARP request. We check to see if the input device is configured for IP forwarding, and the route is a NAT route. The function arp_fwd_proxy is a check to see if we can use proxy ARP for a particular route. Rt points to the route cache entry for this packet.

```
        }  else if (IN_DEV_FORWARD(in_dev)) {
            if ((rt->rt_flags&RTCF_DNAT) ||
                (addr_type == RTN_UNICAST  && rt->u.dst.dev != dev &&
                (arp_fwd_proxy(in_dev, rt) || pneigh_lookup(&arp_tbl,
                  &tip, dev, 0)))) {
                n = neigh_event_ns(&arp_tbl, sha, &sip, dev);
                if (n)
                    neigh_release(n);
```

Now, we timestamp the incoming packet.

```
                if (skb->stamp.tv_sec == 0 ||
                    skb->pkt_type == PACKET_HOST ||
                    in_dev->arp_parms->proxy_delay == 0) {
```

Here we send out the ARP reply packet.

```
                    arp_send(ARPOP_REPLY,ETH_P_ARP,sip,dev,tip,sha,
dev->dev_addr,sha);
                }  else {
```

Pneigh_enqueue puts skb on the proxy queue and starts the timer in the arp_tbl.

```
                    pneigh_enqueue(&arp_tbl, in_dev->arp_parms, skb);
                    in_dev_put(in_dev);
                    return 0;
                }
                goto out;
            }
        }
    }
```

Now we look for an entry in the ARP table matching the address and the network interface device. The zero in the last argument says to not create a new entry.

```
    n = __neigh_lookup(&arp_tbl, &sip, dev, 0);
```

If we are configured for accepting unsolicited ARPs, we process them. Unsolicited ARPs are not accepted by default.

```
#ifdef CONFIG_IP_ACCEPT_UNSOLICITED_ARP
    if (n == NULL &&
        arp->ar_op == htons(ARPOP_REPLY) &&
        inet_addr_type(sip) == RTN_UNICAST)
        n = __neigh_lookup(&arp_tbl, &sip, dev, -1);
#endif
```

If n is not NULL, it means that it found an existing entry in the ARP table.

```
    if (n) {
```

We set the NUD state to reachable. All reachable nodes can have an entry in the ARP cache.

```
        int state = NUD_REACHABLE;
        int override = 0;
```

Override is set to one if we want to override the existing cache entry with a new one. We try to use the first ARP reply if we have received several back to back.

```
        if (jiffies - n->updated >= n->parms->locktime)
            override = 1;
```

If the ARP reply was a broadcast, or the incoming packet is not a reply packet, it means that the NUD state for the destination should not be flagged as reachable. We mark it as stale.

```
        if (arp->ar_op != htons(ARPOP_REPLY) ||
            skb->pkt_type != PACKET_HOST)
            state = NUD_STALE;
```

Finally, we call neigh_update to update the neighbor cache entry. Neigh_release decrements the use count in the neighbor cache entry because we are no longer operating on this entry.

```
        neigh_update(n, sha, state, override, 1);
        neigh_release(n);
    }

out:
    if (in_dev)
        in_dev_put(in_dev);
    kfree_skb(skb);
    return 0;
}
```

9.11 The Internet Control Message Protocol

The ICMP protocol handles many control functions for IPv4. The protocol is often thought of as part of IP, and ICMP messages are carried inside IP packets. There are many different types of ICMP packets, each of which requires different actions. Linux ICMP uses an array to dispatch control functions depending on the input packet types. The array, icmp_control, is defined in the file *linux/net/ipv4/icmp.c*.

```
struct icmp_control {
```

Many ICMP packets require a packet field to be incremented. This structure contains an offset to a statistics counter to be incremented on output and a similar offset for input.

```
    int output_off;
    int input_off;
```

Handler points to a function to handle this specific message type. The next field, error, is set to one if this particular message is an error message.

```
    void (*handler)(struct sk_buff *skb);
    short   error;
} ;
```

Table 9.8 lists the ICMP message types which are defined in file *linux/include/linux/icmp.h*.

Table 9.8: ICMP Messages

Message	**Value**	**Handler**	**Output Counter**	**Output Counter**
ICMP_ECHOREPLY	A00	icmp_discard	IcmpOutEchoReps	IcmpInEchoReps

Table 9.8: ICMP Messages

Message	Value	Handler	Output Counter	Output Counter
ICMP_DEST_UNREACH	A03	icmp_unreach	IcmpOutDestUnreachs	IcmpInDestUnreachs
ICMP_SOURCE_QUENCH	A04	icmp_unreach	IcmpOutSrcQuenchs	IcmpInSrcQuenchs
ICMP_REDIRECT	A05	icmp_redirect	IcmpOutRedirects	IcmpInRedirects
ICMP_ECHO	A08	icmp_echo	IcmpOutEchos	IcmpInEchos
ICMP_TIME_EXCEEDED	11	icmp_unreach	IcmpOutTimeExcds	IcmpInTimeExcds
ICMP_PARAMETERPROB	12	icmp_unreach	IcmpOutParmProbs	IcmpInParmProbs
ICMP_TIMESTAMP	13	icmp_timestamp	IcmpOutTimestamps	IcmpInTimestamps
ICMP_TIMESTAMPREPLY	14	icmp_discard	IcmpOutTimestampReps	IcmpInTimestampReps
ICMP_INFO_REQUEST	15	icmp_discard	dummy	dummy
ICMP_INFO_REPLY	16	icmp_discard	dummy	dummy
ICMP_ADDRESS	17	icmp_address	IcmpOutAddrMasks	IcmpInAddrMasks
ICMP_ADDRESSREPLY	18	icmp_address_reply	IcmpOutAddrMaskReps	IcmpInAddrMaskReps

9.11.1 ICMP Packet Processing

When the AF_INET family is initialized, the initialization function in *linux/net/ipv4/af_inet.c* calls inet_add_protocol to add ICMP to the list of protocols that will receive IPv4 packets after IP is done. The handler function for ICMP is icmp_rcv, defined in file *linux/net/ipv4/icmp.c*. The main purpose of icmp_rcv is to dispatch separate handler functions for all the ICMP message types shown in Table 9.8. It will receive all incoming ICMP packets once they are stripped of their IPv4 headers.

```
int icmp_rcv(struct sk_buff *skb)
{
```

Icmph points to the ICMP header in the incoming packet. Rt is the route cache entry.

```
    struct icmphdr *icmph;
    struct rtable *rt = (struct rtable *)skb->dst;
    ICMP_INC_STATS_BH(IcmpInMsgs);
```

Here, we do some basic packet validity checks. First, we check to if the checksums are correct.

```
    switch (skb->ip_summed) {
    case CHECKSUM_HW:
        if (!(u16)csum_fold(skb->csum))
            break;
        NETDEBUG(if (net_ratelimit())
        printk(KERN_DEBUG "icmp v4 hw csum failure\ n"));
    case CHECKSUM_NONE:
        if ((u16)csum_fold(skb_checksum(skb, 0, skb->len, 0)))
            goto error;
    default:;
    }
```

Now, we check to see that the packet is at least as long as the length of the ICMP header.

```
    if (!pskb_pull(skb, sizeof(struct icmphdr)))
        goto error;
```

If the packet is OK, we set a pointer to the ICMP header in the packet.

```
    icmph = skb->h.icmph;
```

Now, we check to see if we got an illegal ICMP packet type. If so, we silently discard it [RFC 1122].

```
    if (icmph->type > NR_ICMP_TYPES)
        goto error;
     if (rt->rt_flags & (RTCF_BROADCAST | RTCF_MULTICAST)) {
```

An ICMP echo sent to a broadcast address may be silently ignored [RFC 1122]. An ICMP timestamp packet may be silently discarded if it is sent to a broadcast or multicast address. Linux provides a sysctl variable to govern this behavior.

```
        if (icmph->type == ICMP_ECHO &&
            sysctl_icmp_echo_ignore_broadcasts) {
            goto error;
        }
        if (icmph->type != ICMP_ECHO &&
            icmph->type != ICMP_TIMESTAMP &&
            icmph->type != ICMP_ADDRESS &&
            icmph->type != ICMP_ADDRESSREPLY) {
            goto error;
          }
    }
```

Now, we are ready to dispatch one of the control functions listed in Table 9.8 based on the ICMP message type.

```
    ICMP_INC_STATS_BH_FIELD(icmp_pointers[icmph->type].input_off);
    icmp_pointers[icmph->type].handler(skb);
```

Finally, we are done.

```
drop:
    kfree_skb(skb);
    return 0;
error:
    ICMP_INC_STATS_BH(IcmpInErrors);
    goto drop;
}
```

The next function we examine is icmp_unreach, which processes the various types of unreachable of packets.

```
static void icmp_unreach(struct sk_buff *skb)
```

```
. . .
    if (icmph->type == ICMP_DEST_UNREACH) {
```

We decode the code field in the ICMP header to see what type of unreachable packet this is. A few of the codes require special handling.

```
        switch (icmph->code & 15) {
        case ICMP_NET_UNREACH:
        case ICMP_HOST_UNREACH:
        case ICMP_PROT_UNREACH:
        case ICMP_PORT_UNREACH:
            break;
        case ICMP_FRAG_NEEDED:
```

If we get a fragment needed code, we call ip_rt_frag_needed to process the fragment request. This ICMP type is used with the MTU discovery. If we received an ICMP_FRAG_NEEDED packet, it means that we have to decrease our packet size by IP fragmentation.

```
            if (ipv4_config.no_pmtu_disc) {
```

If we are not configured for MTU discovery, we receive this packet type in error.

```
                if (net_ratelimit())
                    printk(KERN_INFO "ICMP: %u.%u.%u.%u: "
                                     "fragmentation needed "
                                     "and DF set.\ n",
                                     NIPQUAD(iph->daddr));
            }  else {
```

We call ip_rt_frag_needed to let IP know that it must fragment the packet.

```
                info = ip_rt_frag_needed(iph,
                                         ntohs(icmph->un.frag.mtu));
                if (!info)
                    goto out;
            }
            break;
```

This packet type indicates that ICMP source routing failed. We don’t support source routing, so we mark an error.

```
        case ICMP_SR_FAILED:
            if (net_ratelimit())
                printk(KERN_INFO "ICMP: %u.%u.%u.%u: Source "
                                 "Route Failed.\ n",
                                 NIPQUAD(iph->daddr));
            break;
        default:
            break;
        }
```

We check for packets of illegal codes. If so, we silently discard them.

```
        if (icmph->code > NR_ICMP_UNREACH)
            goto out;
    }  else if (icmph->type == ICMP_PARAMETERPROB)
        info = ntohl(icmph->un.gateway) >> 24;
. . .
```

We must pass an ICMP “parameter problem” packet up through the protocol layers and eventually back to the open socket [RFC1122]. We also pass the code up to any open raw socket, but that code snippet isn’t shown here. Hash is calculated form the protocol field in the IP header and is used to find the error socket for the protocol.

```
    rcu_read_lock();
    ipprot = inet_protos[hash];
    smp_read_barrier_depends();
```

Here we call the registered error handler function for the transport protocol indicated in the protocol field so that protocol can be informed of the error condition.

```
    if (ipprot && ipprot->err_handler)
        ipprot->err_handler(skb, info);
    rcu_read_unlock();

out:
    return;
out_err:
    ICMP_INC_STATS_BH(IcmpInErrors);
    goto out;
}
```

The other functions shown in Table 9.8 are called in the same way. For example, the function icmp_redirect is called when an ICMP redirect packet is received.

```
static void icmp_redirect(struct sk_buff *skb)
```

This function gets the header of the packet that caused the redirect and calls the function ip_rt_redirect to update the routing table.

The next function, icmp_echo, is called when pings, ICMP_ECHO type packets, are received.

```
static void icmp_echo(struct sk_buff *skb);
```

This function, icmp_timestamp, is the handler for ICMP timestamp requests. It changes the ICMP header type field to ICMP_ECHOREPLAY and calls icmp_reply to send the reply packet [RFC 1122].

```
static void icmp_timestamp(struct sk_buff *skb);
```

It builds a packet of type ICMP_TIMESTAMPREPLY, calculates the time in the correct format, and sends the packet by calling icmp_reply.

9.11.2 Sending ICMP Packets

The function icmp_reply builds and sends ICMP response messages of various types.

```
static void icmp_reply(struct icmp_bxm *icmp_param, struct sk_buff *skb)
{
```

This function does a few interesting things before sending the packet. Later, it builds a routing table entry, actually the destination cache entry. It needs a pointer to the dst_entry structure in the output socket buffer, skb.

```
    struct rtable *rt = (struct rtable *)skb->dst;
    if (ip_options_echo(&icmp_param->replyopts, skb))
        goto out;

    if (icmp_xmit_lock())
        return;

    icmp_param->data.icmph.checksum = 0;
    icmp_out_count(icmp_param->data.icmph.type);

    inet->tos = skb->nh.iph->tos;
    daddr = ipc.addr = rt->rt_src;
    ipc.opt = NULL;
    if (icmp_param->replyopts.optlen) {
        ipc.opt = &icmp_param->replyopts;
        if (ipc.opt->srr)
            daddr = icmp_param->replyopts.faddr;
    }
```

We build a route cache lookup flowi structure, and get a route cache entry. Then it checks to see if the ICMP output packets are rate limited and if we should allow this packet to go out.

```
    {
        struct flowi fl = {  .nl_u = {  .ip4_u =
                                        {  .daddr = daddr,
                                           .saddr = rt->rt_spec_dst,
                                           .tos = RT_TOS(skb->nh.iph->tos)
}  } ,
                             .proto = IPPROTO_ICMP } ;
        if (ip_route_output_key(&rt, &fl))
            goto out_unlock;
```

This is to check to see if we are doing ICMP rate limiting. ICMP rate limiting is a technique to prevent DoS attacks by flooding a system with ICMP packets.

```
    if (icmpv4_xrlim_allow(rt, icmp_param->data.icmph.type,
                           icmp_param->data.icmph.code))
```

We queue the packet for transmission, release our route cache entry, and go out.

```
        icmp_push_reply(icmp_param, &ipc, rt);
    ip_rt_put(rt);
```

```
out_unlock:
    icmp_xmit_unlock();
out:;
}
```

A socket is created when the ICMP protocol was initialized. In this function, we use the socket as a way to transmit output packets down the protocol stack. The packets are queued for transmission by placing them on the socket's output queue. The function icmp_push_reply queues up completed ICMP packets to a socket for transmission.

```
static void icmp_push_reply(struct icmp_bxm *icmp_param,
                            struct ipcm_cookie *ipc, struct rtable *rt)
{
    struct sk_buff *skb;
```

Here, we append data to the socket. We make sure there is room on the sockets write_queue. We calculate the checksum as we copy the data.

```
    ip_append_data(icmp_socket->sk, icmp_glue_bits, icmp_param,
                   icmp_param->data_len+icmp_param->head_len,
                   icmp_param->head_len,
                   ipc, rt, MSG_DONTWAIT);
    if ((skb = skb_peek(&icmp_socket->sk->sk_write_queue)) != NULL) {
        struct icmphdr *icmph = skb->h.icmph;
        unsigned int csum = 0;
        struct sk_buff *skb1;
        skb_queue_walk(&icmp_socket->sk->sk_write_queue, skb1) {
            csum = csum_add(csum, skb1->csum);
        }
        csum = csum_partial_copy_nocheck((void *)&icmp_param->data,
                                         (char *)icmph,
                                         icmp_param->head_len, csum);
        icmph->checksum = csum_fold(csum);
        skb->ip_summed = CHECKSUM_NONE;
```

We call ip_push_pending_frames to force IP to send the packets.

```
        ip_push_pending_frames(icmp_socket->sk);
    }
}
```

9.12 Multicast and IGMP

The Internet Group Management Protocol (IGMP) is for exchanging messages to manage multicast routing and message transmission. Essentially, the purpose of IGMP is to associate a group of unicast addresses to a specific class D multicast address. The protocol exchanges information about these groups between hosts and routers.

There are three versions of IGMP: version 1 [RFC 1112], version 2, and version 3 [RFC 3376]. All three versions are interoperable. Version 1 specifies two types of messages, a membership query message sent by IGMP routers, and a membership report sent by IGMP hosts. A host

sends the membership report to tell multicast routers to forward messages that are sent to the group address. Version 2 adds a leave group message so a host can tell a router when it wants to leave a group and no longer receive packets sent to that particular group address. Version 3 adds source-specific information. Linux supports the host side of IGMP version 3.

In Linux, multicast capability is optional. It can be completely deconfigured from the Linux kernel. Although the Linux TCP/IP implementation supports host-side multicasting, the kernel by itself is not a multicast router. The routing engine "knows" how to forward multicast messages, but Linux does not support the automatic updating of the routing tables by incoming IGMP reports. An application layer program such as *mrouted* is required to fully implement multicast routing in Linux. All the functions described in this section are implemented in the file *linux/net/ipv4/igmp.c*.

There are two important data structures used by Linux IGMP. The first of these is the multicast request, ip_mreqn, defined in file *linux/include/linux/in.h*. It is for communicating join- or leave-group requests between the socket layer and the internal kernel functions that process the requests.

```
struct ip_mreqn
{
```

The first field in this structure is the multicast address for the group that we want to either join or leave.

```
    struct in_addr      imr_multiaddr;
```

This field specifies the IP address of the local interface through which we want to receive datagrams sent to the multicast group address.

```
    struct in_addr      imr_address;
```

This field is the index of the interface through which we want to receive the datagrams.

```
    int           imr_ifindex;
} ;
```

The second data structure we want to consider is the multicast socket list, ip_mc_socklist, defined in *linux/include/linux/igmp.h*. It is for holding the list of addresses for membership reports.

```
struct ip_mc_socklist
{
    struct ip_mc_socklist     *next;
    int                 count;
    struct ip_mreqn     multi;
```

The next field, sfmode, is the multicast source filter mode. It says whether to include or exclude this entry.

```
    unsigned int        sfmode;
```

The next field is for multicast source filtering.

```
    struct ip_sf_socklist*sflist;
} ;
```

9.12.1 Receiving IGMP Packets

As is the case with the other member protocols in the AF_INET family, the handler function for IGMP is registered with the IP protocol to receive IP packets for each protocol type. This is done during the initialization process by the function inet_init in the file *linux/net/ipv4/af_inet.c*. The IGMP main receive function registered for IGMP is igmp_rcv.

```
int igmp_rcv(struct sk_buff *skb)
    struct igmphdr *ih;
```

We need to get the in_device structure from the network interface device, because this is where the multicast address lists are stored. If in_dev is NULL, that means that the network interface can't receive or transmit IP packets.

```
    struct in_device *in_dev = in_dev_get(skb->dev);
    int len = skb->len;
    if (in_dev==NULL) {
        kfree_skb(skb);
        return 0;
    }
```

Here we check to make sure that the incoming packet is at least long enough to hold the IGMP header information.

```
    if (!pskb_may_pull(skb, sizeof(struct igmphdr)) ||
        (u16)csum_fold(skb_checksum(skb, 0, len, 0))) {
        in_dev_put(in_dev);
        kfree_skb(skb);
        return 0;
    }
    ih = skb->h.igmph;
```

The type field is retrieved from the IGMP header.

```
    switch (ih->type) {
    case IGMP_HOST_MEMBERSHIP_QUERY:
```

The function igmp_heard_query handles incoming IGMP queries.

```
        igmp_heard_query(in_dev, skb, len);
        break;
    case IGMP_HOST_MEMBERSHIP_REPORT:
    case IGMPV2_HOST_MEMBERSHIP_REPORT:
    case IGMPV3_HOST_MEMBERSHIP_REPORT:
```

We check to see that we are not looking at our own looped back report.

```
        if (((struct rtable*)skb->dst)->fl.iif == 0)
            break;
```

The function igmp_heard_report handles incoming reports.

```
        igmp_heard_report(in_dev, ih->group);
        break;
```

The following message type is only for the PIM protocol. The next series of IGMP message types are not supported.

```
    case IGMP_PIM:
#ifdef CONFIG_IP_PIMSM_V1
        in_dev_put(in_dev);
        return pim_rcv_v1(skb);
#endif
    case IGMP_DVMRP:
    case IGMP_TRACE:
case IGMP_HOST_LEAVE_MESSAGE:
    case IGMP_MTRACE:
    case IGMP_MTRACE_RESP:
        break;
    default:
        NETDEBUG(printk(KERN_DEBUG "New IGMP type=%d,
why we do not know about it?\ n", ih->type));
    }
    in_dev_put(in_dev);
    kfree_skb(skb);
    return 0;
}
```

9.12.2 Handling IGMP Queries

IGMP queries are sent from routers to the all-hosts. Hosts respond to the queries with membership reports. The function igmp_heard_query processes incoming IGMP queries.

```
static void igmp_heard_query(struct in_device *in_dev, struct sk_buff *skb,
                             int len)
{
```

Ih points to the IGMP header.

```
    struct igmphdr            *ih = skb->h.igmph;
    struct igmpv3_query *ih3 = (struct igmpv3_query *)ih;
    struct ip_mc_list   *im;
```

Group is the multicasting group (class D) address.

```
    u32                group = ih->group;
    int                max_delay;
    int                mark = 0;
```

The first thing we have to do is figure out if the incoming query is version 1, 2, or 3. Reports are sent out at pseudo-random intervals. We set timers for version 1 and version 2. A packet length of eight means that we have either a version 1 or a version 2 query.

```
    if (len == 8) {
        if (ih->code == 0) {
```

A zero value in code means it is a version 1 query.

```
            max_delay = IGMP_Query_Response_Interval;
            in_dev->mr_v1_seen = jiffies +
                IGMP_V1_Router_Present_Timeout;
            group = 0;
        }  else {
            max_delay = ih->code*(HZ/IGMP_TIMER_SCALE);
            in_dev->mr_v2_seen = jiffies +
                IGMP_V2_Router_Present_Timeout;
        }
```

Here, we cancel the interface change timer.

```
        in_dev->mr_ifc_count = 0;
        if (del_timer(&in_dev->mr_ifc_timer))
            __in_dev_put(in_dev);
```

Clear all deleted report items.

```
        igmpv3_clear_delrec(in_dev);
    }  else if (len < 12) {
```

If the length is greater than 8 but less than 12, this must be a bogus packet. The caller frees it.

```
        return;
    }  else {
```

The incoming packet must be a version 3 query. We make sure the packet is at least as long as the v3 query header.

```
        if (!pskb_may_pull(skb, sizeof(struct igmpv3_query)))
            return;
```

The variable ih3 points to the version 3 query header. We check that the packet is sufficiently long to hold the source addresses if there are any.

```
        ih3 = (struct igmpv3_query *) skb->h.raw;
```

We check the number of source addresses in the v3 packet.

```
        if (ih3->nsrcs) {
            if (!pskb_may_pull(skb, sizeof(struct igmpv3_query)
                               + ntohs(ih3->nsrcs)*sizeof(__u32)))
                return;
```

```
            ih3 = (struct igmpv3_query *) skb->h.raw;
        }
        max_delay = IGMPV3_MRC(ih3->code)*(HZ/IGMP_TIMER_SCALE);
        if (!max_delay)
            max_delay = 1;
        in_dev->mr_maxdelay = max_delay;
        if (ih3->qrv)
            in_dev->mr_qrv = ih3->qrv;
```

The value in the IGMP group field lets us know if this is a general query or whether it is a source-specific query.

```
        if (!group) {
```

Number-of-sources must be zero for a general query.

```
            if (ih3->nsrcs)
                return
```

We start the group query timer. Each interface has a timer, which is in the inet_device structure described in Chapter 6.

```
            igmp_gq_start_timer(in_dev);
            return;
        }
```

If there are included sources, we indicate that by setting mark.

```
        mark = ih3->nsrcs != 0;
    }
```

Now, we start timers in all of the membership groups to which the query applies for the interface from which we received the query. If a timer is already running, it is reset. We don't start timers for any well-known groups such as the all hosts group, 224.0.0.1.

```
    read_lock(&in_dev->lock);
    for (im=in_dev->mc_list; im!=NULL; im=im->next) {
        if (group && group != im->multiaddr)
            continue;
        if (im->multiaddr == IGMP_ALL_HOSTS)
            continue;
        spin_lock_bh(&im->lock);
        if (im->tm_running)
            im->gsquery = im->gsquery && mark;
        else
            im->gsquery = mark;
        if (im->gsquery)
            igmp_marksources(im, ntohs(ih3->nsrcs), ih3->srcs);
        spin_unlock_bh(&im->lock);
        igmp_mod_timer(im, max_delay);
    }
    read_unlock(&in_dev->lock);
}
```

9.12.3 Handling IGMP Reports

IGMP reports are handled by an IGMP router. Although Linux receives the reports, it depends on a multicast routing daemon such as *mrouted* to interpret the raw IGMP packets and update the routing tables.

This function listens for IGMP reports, igmp_heard_report.

```
static void igmp_heard_report(struct in_device *in_dev, u32 group);
```

Igmp_heard report does little more than stop the timer for the group in the membership report.

9.12.4 Adding and Leaving Groups

When an application using IGMP version 2 wants to join a group, it calls the setsockopt function with the option IP_ADD_MEMBERSHIP or MCAST_JOIN_GROUP. If setsockopt sees either of these options, it calls ip_mc_join_group.

```
int ip_mc_join_group(struct sock *sk , struct ip_mreqn *imr);
```

This function must find the network interface specified by the index in ip_mreqn. Next, it checks for a match between the multicast address in ip_mreqn and the list of multicast addresses assigned to the interface. If it finds a match, it increments the reference count and adds the unicast address to the group. Then, it calls ip_mc_inc_group, which will set the timer for that group. Later when the timer expires, the group report will be transmitted.

If the application using IGMP version 2 is requesting to leave a group, it will call setsockopt with either the MCAST_LEAVE_GROUP or the IP_DROP_MEMBERSHIP option set. In this case, setsockopt will call the function ip_mc_leave_group.

```
int ip_mc_leave_group(struct sock *sk, struct ip_mreqn *imr);
```

As with the join group function, the second argument, imr, points to the multicast request structure shown in Section 9.12. The function must find the inet_device for the interface. It then removes the specified multicast address from the interface.

9.13 Sending Packets from IP

In Chapter 6, we explained that IP queues its packet to the output network interface drivers through the queuing layer. As discussed previously, Linux provides multiple queuing disciplines that are selected by traffic class. When IP is done forming a packet, it places the packet on one of these queues. However, the IP output routine is not called directly. Instead, it is done indirectly through the output operation field in the neighbor cache entry.

In general, all internal packet routing is done through the neighbor cache. For example, if a packet is supposed to be forwarded, the neighbor cache output pointer will point to ip_forward. Once all the processing of an output packet is complete, and if there is an unresolved route for the packet, the neighbor cache's output operation may point to the function, ip_output. If there is

a resolved route for the packet, the output function will point to dev_queue_xmit. The queuing layer and the lower part of the IP transmit facility was discussed in Chapter 6.

In this section, we discuss the functions called directly from the upper layers to construct an IP datagram from a UDP datagram or a TCP segment. IP provides several functions for this purpose, and they are defined in the file *linux/net/ipv4/ip_output.c*. Some of the functions we will discuss are used by datagram- oriented transport layer protocols such as UDP or raw sockets. One of the functions builds a single IP datagram from pieces. Another does the same thing, but builds the datagram from mapped memory pages. In addition, we will discuss a function used primarily by TCP to make an IP datagram from a TCP segment.

The next function we will discuss is ip_append_data, which is called from raw, UDP, or other upper layer protocols. It constructs a single IP datagram from various pieces of data.

```
int ip_append_data(struct sock *sk,
int getfrag(void *from, char *to, int offset, int len, int odd,
            struct sk_buff *skb),
void *from,  int length,
            int transhdrlen,  struct ipcm_cookie *ipc,
            struct rtable *rt, unsigned int flags);
```

Getfrag points to the function called to get the IP fragment contents. Usually, it will point to ip_generic_getfrag, which copies the actual data from user space into the kernel buffers while simultaneously calculating the IP checksum. The argument from points to the data buffer, and offset is an offset into the data buffer. Transhdrlen is the header length without IP options. The parameter ipc points to the control message structure from the sendmsg socket call. This contains the IP options to be put in the IP header. The route cache entry for the output packet is in rt, and this is where we get the destination address.

The first thing that we do in ip_append_data is check to see if the socket write queue is empty. If so, it builds the corked data that will become the IP header of each outgoing fragment sent through this socket. It calculates a fragment length based on the MTU and the IP header size. Next, it builds a chain of sk_buffs and places them on the socket's output queue. If the output device has scatter-gather capability, the output packet is set up as a chain of mapped pages ready for transmission by the network interface hardware. For each socket buffer, we call __skb_queue_tail to place the skb on the socket's pending write queue.

The next function, ip_append_page, conceptually does the same thing as ip_append_data. It is much simpler, though, because the data to assemble into IP output packets is already in a list of mapped pages.

```
ssize_t     ip_append_page(struct sock *sk, struct page
*page, int offset, size_t size, int flags)
```

Typically, this function is called by the upper layer's sendpage function such as udp_sendpage, defined in file *linux/net/ipv4/udp.c*. It allocates a sk_buff if there is room on the socket's write queue. The skb holds the IP header. Then, the pages in the list are appended to the skb. Finally, the function __skb_queue_tail is called to place the skb on the sockets pending write queue.

Another function is provided so streaming transport protocols such as TCP can efficiently send segments through IP. This function is ip_queue_xmit.

```
int ip_queue_xmit(struct sk_buff *skb, int ipfragok)
```

The first thing this function does is check to see if there is already a route for the output packet. We do this by checking dst field in the sk_buff, which points to the route cache entry. If there is no route, we build a flow information structure from the information in skb such as the destination address, port, and ToS value. Then, we call ip_route_output_flow to resolve the route. Once we have a route, we construct the IP header, calculate the IP checksum, and call the function pointed to by the output field in the route cache entry, which generally points to ip_forward or ip_output depending on the route.

9.14 Receiving Packets in IP

When a packet arrives in IP, a few basic decisions must be made. First, IP must decide whether the packet is a valid IP packet or whether to discard it. The next decision is either to forward the packet out an interface or send it to an internal destination. If it is sent out another interface, IP must figure out who to send the packet to, and whether it is sent to the final destination or to a gateway. Finally, for packets it is keeping, IP must decide which higher-level protocol should receive the packet next. The first decision is based on the basic packet integrity. The second decision, to forward or not, is based on a search of the routing tables. Section 9.3 discusses the IP routing mechanism and how forwarding is done based on a search of the routing tables. In this section, we start with an IP packet as it arrives in the IP receive packet handler and follow it as it is processed. Figure 9.1 shows the processing sequence for packets received by the IP receive packet handler.

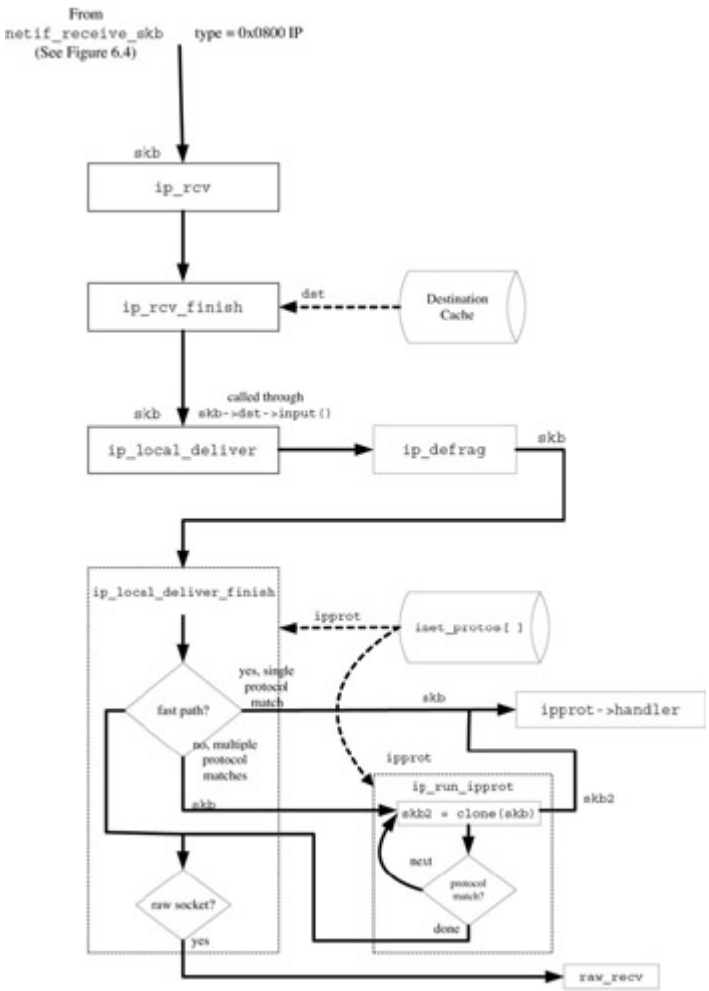

Figure 9.1: Receive packet sequence.

Chapter 6, Section 6.6 covers how packets are removed from the input packet queue and how the NET_RX_SOFTIRQ tasklet checks the link layer header type field to determine which packet handler to call for the particular type. In the case of the IP protocol, the packet type is 0x0800.

9.14.1 The IP Input Function ip_rcv

As discussed in Section 9.2, IP registers the packet handler function ip_rcv during its initialization phase. Ip_rcv is the main input function for the IP protocol and it is passed the incoming packet in a sk_buff, a pointer to the incoming network interface, dev, and a pointer to the packet type, pt.

```
int ip_rcv(struct sk_buff *skb, struct net_device *dev,
           struct packet_type *pt)
{
```

First, we get a pointer to the IP packet header, iph.

```
    struct iphdr *iph;
```

The pkt_type field of the socket buffer, skb, indicates the class of the packet. Later, we check this field to see if the packet was not sent to this host. Since we are currently executing a receive routine for this host, if the packet was not sent to us, we drop it. This is really a redundant check. Unless the packet was received promiscuously, it wouldn't have been sent to the IP receive routine if it was not intended for our consumption. Values for the packet type field are shown in Chapter 7.

```
if (skb->pkt_type . PACKET_OTHERHOST)
    goto drop;

IP_INC_STATS_BH(IpInReceives);

if ((skb = skb_share_check(skb, GFP_ATOMIC)) == NULL)
    goto out;
```

This is a check to see if the packet is at least as long as the IP header, and if not, we discard it.

```
if (!pskb_may_pull(skb, sizeof(struct iphdr)))
    goto inhdr_error;

iph = skb->nh.iph;
```

Now we check the header checksum to see if the IP version is four and that the header length is at least 20 bytes. If not, we silently discard the packet [RFC 1122]. We check to see that the packet is big enough to hold the header and any IP options.

```
if (iph->ihl < 5 || iph->version != 4)
    goto inhdr_error;
if (!pskb_may_pull(skb, iph->ihl*4))
    goto inhdr_error;
```

We get a pointer to the header and calculate the IP header checksum.

```
iph = skb->nh.iph;
if (ip_fast_csum((u8 *)iph, iph->ihl) != 0)
    goto inhdr_error;

{
    __u32 len = ntohs(iph->tot_len);
    if (skb->len < len || len < (iph->ihl<<2))
        goto inhdr_error;
```

Since the network interface may have padded the end of the buffer holding the packet, __pskb_trim will eliminate the padding so the length of the skb is equal to the packet length field in the IP header.

```
    if (skb->len > len) {
        __pskb_trim(skb, len);
        if (skb->ip_summed == CHECKSUM_HW)
            skb->ip_summed = CHECKSUM_NONE;
    }
}
```

Ip_rcv_finish continues the input packet processing, described in the next section.

```
    return NF_HOOK(PF_INET, NF_IP_PRE_ROUTING, skb, dev, NULL,
                   ip_rcv_finish);

inhdr_error:
    IP_INC_STATS_BH(IpInHdrErrors);
drop:
            kfree_skb(skb);
out:
        return NET_RX_DROP;
}
```

9.14.2 Continue IP Input Processing, ip_rcv_finish

Ip_rcv_finish is defined in file *linux/net/ipv4/ip_output.c*. It is called by ip_rcv to continue processing the input packet, skb. At the point where we arrive in this function, we have done some basic input packet filtering. At this point, the main purpose of the input packet processing is to determine the processing path for the input packet. We determine where the packet will be sent next. This is not just for external packet routing; decisions must be made for packets intended for internal processing, too. After we know what is to be done with the input packet, Ip_rcv_finish updates the destination information for the packet by setting the dst field of the skb to point to an entry in the destination cache. Ip_rcv_finish also extracts the IP options from the input packet (if there are any) and processes them.

```
static inline int ip_rcv_finish(struct sk_buff *skb)
{
    struct net_device *dev = skb->dev;
    struct iphdr *iph = skb->nh.iph;
```

Here we set the destination cache (dst field of the skb) for this packet if it is not already set. We do this by calling ip_route_input, which determines the internal route and sets up dst to point to the entry in the destination or routing cache.

```
    if (skb->dst == NULL) {
        if (ip_route_input(skb, iph->daddr, iph->saddr, iph->tos, dev))
            goto drop;
    }
```

This section of code is for traffic class routing.

```
#ifdef CONFIG_NET_CLS_ROUTE
    if (skb->dst->tclassid) {
        struct ip_rt_acct *st = ip_rt_acct + 256*smp_processor_id();
        u32 idx = skb->dst->tclassid;
        st[idx&0xFF].o_packets++;
        st[idx&0xFF].o_bytes+=skb->len;
        st[(idx>>16)&0xFF].i_packets++;
        st[(idx>>16)&0xFF].i_bytes+=skb->len;
    }
#endif
```

Next, we process the IP options if there are any in this incoming packet. We determine if we have IP options by looking to see if the IP header length is greater than the normal length, 20 bytes. The IP header length field indicates the number of 32-bit words, so if it is greater than five, we assume that options are present in the packet. If there are IP options, skb_cow is called to make sure the skb is unshared and OK for writing. This step is done because it is possible that the buffer contents could change while processing some of the IP options, and we must make sure that the buffer is safe for modifying.

```
    if (iph->ihl > 5) {
        struct ip_options *opt;
        if (skb_cow(skb, skb_headroom(skb)))
            goto drop;
        iph = skb->nh.iph;
```

Ip_options_compile gathers up the options from the input packet and fills in the ip_options structure, pointed to by the opt field of the IP control buffer. If it fails, that means that the packet had a bad option.

```
        if (ip_options_compile(NULL, skb))
            goto inhdr_error;
```

Here we check to see if the IP source routing option is present in this packet. We retrieve the pointer to the in_device structure from the input network in terface, dev. In_device is a pseudo-device structure used to keep all address lists associated with the network interface. If this incoming packet contains the source routing option but the input device is not configured for source routing, we consider the input packet to be a "Martian" and drop it. Otherwise, we process the source routing option. Ip_options_rcv_srr does the processing of the option and returns an error if unsuccessful and the packet is dropped. Otherwise, we proceed as if it was a normal packet. See Stevens, Chapter 8, Section 8.5 [STEV94] for a discussion of the IP source routing option.

```
        opt = &(IPCB(skb)->opt);
        if (opt->srr) {
            struct in_device *in_dev = in_dev_get(dev);
            if (in_dev) {
                if (!IN_DEV_SOURCE_ROUTE(in_dev)) {
                    if (IN_DEV_LOG_MARTIANS(in_dev) && net_ratelimit())
                        printk(KERN_INFO
                          "source route option %u.%u.%u.%u -> %u.%u.%u.%u\ n",
                          NIPQUAD(iph->saddr), NIPQUAD(iph->daddr));
                    in_dev_put(in_dev);
                    goto drop;
                }
                in_dev_put(in_dev);
            }
            if (ip_options_rcv_srr(skb))
                goto drop;
        }
    }
```

If the incoming packet passed all the initial checks, we pass it on to the next processing step by calling the input function pointer defined for the destination cache in the socket buffer, dst. The

value of the dst field was determined earlier when ip_route_input was called. For incoming packets intended for any of the IP family protocols such as UDP, TCP, or IGMP, the input operations field in dst will point to the function, ip_local_deliver.

```
    return dst_input(skb);
inhdr_error:
    IP_INC_STATS_BH(IpInHdrErrors);
drop:
        kfree_skb(skb);
        return NET_RX_DROP;
}
```

9.14.3 Delivering Input IP Packets to Higher-level Protocols

After ip_rcv_finish is done with the incoming packets, if the packets are for local consumption and we have a local destination within this host, the packets will be passed to the function ip_local_deliver. This function will decide which higher-layer protocols are to receive the packet next. Ip_local_deliver is passed a pointer to the input packet in skb.

```
int ip_local_deliver(struct sk_buff *skb)
{
```

ip_defrag is called to reassemble IP fragments. The reassembled IP packet is returned in skb.

```
    if (skb->nh.iph->frag_off & htons(IP_MF|IP_OFFSET)) {
        skb = ip_defrag(skb);
        if (!skb)
            return 0;
    }
```

Once the IP packet is complete, we are ready to hand off the packets to the higher-level protocols.

```
    return NF_HOOK(PF_INET, NF_IP_LOCAL_IN, skb, skb->dev, NULL,
                   ip_local_deliver_finish);
}
```

The function ip_local_deliver_finish delivers the input packet to all the protocols that will receive the packet next. It determines if the input packets are to be sent to a single higher-level protocol or to multiple protocols. IP can dispatch packets to multiple protocols under several conditions. For example, if any protocol has asked to receive packets promiscuously, it must receive a copy (actually a clone) of the input packet. If there are one or more raw sockets open, each of the raw sockets will receive a clone of the input packet.

```
static inline int ip_local_deliver_finish(struct sk_buff *skb)
{
```

Ihl is the length of the IP header.

```
    int ihl = skb->nh.iph->ihl*4;
```

The following code is run if net filters are configured into the Linux kernel.

```
#ifdef CONFIG_NETFILTER_DEBUG
    nf_debug_ip_local_deliver(skb);
#endif /*CONFIG_NETFILTER_DEBUG*/
```

At this point, we have consumed the IP header, so we call __skb_pull to set the data field in the skb to point to the IP packet payload.

```
    __skb_pull(skb, ihl);
#ifdef CONFIG_NETFILTER
    nf_conntrack_put(skb->nfct);
    skb->nfct = NULL;
#endif /*CONFIG_NETFILTER*/
```

The raw header field in the skb points to the first byte after the IP header.

```
         skb->h.raw = skb->data;
    rcu_read_lock();
    {
```

We retrieve the protocol field from the IP header.

```
        int protocol = skb->nh.iph->protocol;
```

Hash is used as an index into both the inet_protos and raw_v4_htable protocol tables. Hash is set from the 8-bit protocol field from the IP header. Both raw_v4_htable (the raw hash table) and inet_protos are sized large enough to hold all the common protocols defined for IP [IAPROT03]. The size of both these arrays is defined by the constant, MAX_INET_PROTOS, equal to 32. There isn't a separate slot for every single protocol, and the inet_protocol structure pointers are put on a list if the hash lookup generates a conflict. However, the protocol hashing provides a fast decision for the common protocols in TCP/IP, UDP, TCP, IGMP, and ICMP. Chapter 6, Section 6.7 discusses how the member protocols of the AF_INET family register their handler functions using the registration facility.

Raw_sk is a pointer to a sock structure for a raw socket. This will be used to point to a slot in the raw socket hash table, raw_v4_htable. Each location in the table contains either a pointer to a sock structure if there is an open raw socket for this protocol number, or NULL. Hash indexes into the raw socket array to yield a pointer to an open raw socket, or NULL if there isn't one.

```
        int hash;
        struct sock *raw_sk;
        struct inet_protocol *ipprot;
resubmit:
```

If there is a raw socket for protocol, we will call raw_v4_input to distribute clones of skb to each of the raw sockets.

```
        hash = protocol & (MAX_INET_PROTOS - 1);
        raw_sk = sk_head(&raw_v4_htable[hash]);

        if(raw_sk)
            raw_sk = raw_v4_input(skb, skb->nh.iph, hash);
```

Next, the inet_protocol array, inet_protos, is indexed by hash and ipprot is set to the first matching protocol.

```
        if ((ipprot = inet_protos[hash]) != NULL) {
            int ret;
            smp_read_barrier_depends();
            if (!ipprot->no_policy &&
                !xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb)) {
                kfree_skb(skb);
                goto out;
            }
```

Here is where we call the handler function for the upper layer protocol.

```
            ret = ipprot->handler(skb);
            if (ret < 0) {
                protocol = -ret;
                goto resubmit;
            }
            IP_INC_STATS_BH(IpInDelivers);
        }  else {
```

If we didn't have a raw socket, we check for unknown protocols. If the destination protocol is unknown, we send an ICMP unreachable message.

```
            if (!raw_sk) {
                if (xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb)) {
                    IP_INC_STATS_BH(IpInUnknownProtos);
                    icmp_send(skb, ICMP_DEST_UNREACH,

                                       ICMP_PROT_UNREACH, 0);
                }
            }  else
                IP_INC_STATS_BH(IpInDelivers);
            kfree_skb(skb);
        }
    }
 out:
    rcu_read_unlock();

    return 0;
}
```

9.15 Summary

In this chapter, we covered the IPv4 protocol. Essentially all internal packet routing in IP is through the neighbor system, which includes the destination cache. Although the framework for the routing cache was discussed in Chapter 6, in this chapter we discussed the routing policy database and the FIB as well as how the IPv4-specific route cache is implemented. We followed input packets as they arrived in IP and were delivered to higher-level protocols. We also discussed what happens when the higher-level protocols want to transmit a packet through IP.

We covered how packets are routed and described the main route resolver functions for input packets and output packets. We also discussed the ICMP and IGMP protocols.

Chapter 10: Receiving Data in the Transport Layer, UDP, and TCP

This chapter continues to examine the implementation of the TCP/IP transport layer protocols in Linux. In the previous two chapters, we followed the data as it was written into a socket, passed through the transport layer, and transmitted by the network layer. In this chapter, we will see what happens as data is received in the transport layer from the network layer. We discuss both the UDP and TCP transport layer protocols.

10.1 Introduction

In this chapter, we show what happens when a packet arrives in the transport layer. We cover the UDP protocol, which processes packets for SOCK_DGRAM type sockets, and the TCP protocol, which processes packets for SOCK_STREAM type sockets. As we saw in Chapter 8, "Sending the Data from the Socket through UDP and TCP," UDP is far simpler than TCP because it doesn't handle any state information, so the majority of this chapter is devoted to TCP. However, in this chapter, we discuss the packet handling for both protocols. Since TCP is far more complicated than UDP, the discussion includes handling of the queues and processing of input TCP segments in the various states. In general, we continue the discussion of earlier chapters by showing how each transport layer protocol works by following the data as it flows up from the network layer, IP through the transport layer, and up into the receiving socket. The flow of input packets is straightforward for UDP and can be discussed separately from the flow of output packets in Chapter 8. This is because UDP is limited to individual packet processing and doesn't have to maintain as much internal state information. However, with TCP, since the received packet flow depends on the internal state of the protocol, it isn't possible to completely separate the discussion of the receive side from the send side. Therefore, the TCP-related sections in this chapter should be read while referring to Chapter 8.

As is the case with the send side, covered earlier in Chapter 8, all the user-level read functions are converted into a single function at the socket layer. In UDP, the function doing the work is udp_recvmsg, and for TCP, it is tcp_recvmsg. The bulk of this chapter covers these two functions and other associated processing. UDP is a simpler case because packet reception in UDP essentially consists of little more than checksum calculation and copying of the packet data form kernel space to user space. Sections 10.3 and 10.4 cover the UDP-specific receive packet handling. TCP is far more complex, and the receive side must ensure coordination with the send-side processing, TCP state handling, and keeping track of acknowledgments. Sections 10.5 through 10.9 cover the processing of received packets in TCP.

10.2 Receive-Side Packet Handling

The transport layer protocols—TCP, UDP, and other member protocols in the AF_INET protocol family—all receive packets form the network layer, IP. As we saw in Chapter 9, "The Network Layer, IP," when IP is done processing an input packet it dispatches the packet to a higher-layer protocol's receive function by decoding the 1-byte protocol field in the IP header. It

uses a hash table to determine whether TCP, UDP, or some other protocol should receive the packet.

In Chapter 6, "The Linux TCP/IP Stack," we discussed the hash tables, the inet_protocol structure, and how incoming IP packets are de-multiplexed. In Chapter 9, we discussed the flow of incoming packets through the IP input routine. In this chapter, we will start following the packets as soon as they arrive in each of the two transport layer protocols' handler functions.

10.3 Receiving a Packet in UDP

In Chapter 6, we saw how protocols register with the AF_INET family by calling the inet_add_protocol function in *linux/net/ipv4/protocol.c*. This initialization step happens when the AF_INET initialization function, inet_init, in the file *linux/net/ipv4/af_inet.c*, runs at kernel startup time. In the inet_protocol structure for UDP, the value in the protocol field is IPPPROTO_UDP, and the function defined in the handler field is udp_rcv. This is the function that gets called for all incoming UDP packets passed up to us by IP.

10.3.1 UDP Receive Handler Function, udp_rcv

Udp_rcv in file *linux/net/ipv4/udp.c* is the first function in UDP that sees a UDP input packet after IP is done with it. This function executed when the protocol field in the IP header is IPPPROTO_UDP, or the value 17.

```
int udp_rcv(struct sk_buff *skb)
{
```

The variable, sk, will point to the socket that gets this packet if there is a socket open on this port. Uh points to the UDP header. Rt gets the routing table entry from the destination cache.

```
    struct sock *sk;
    struct udphdr *uh;
    unsigned short ulen;
    struct rtable *rt = (struct rtable*)skb->dst;
    u32 saddr = skb->nh.iph->saddr;
    u32 daddr = skb->nh.iph->daddr;
    int len = skb->len;
    IP_INC_STATS_BH(IpInDelivers);
```

The function pskb_may_pull is called to confirm that there is sufficient space in the socket buffer to hold the UDP header. Uh points to the UDP header, and ulen is the value of the length field in the UDP header.

```
    if (!pskb_may_pull(skb, sizeof(struct udphdr)))
        goto no_header;
    uh = skb->h.uh;
    ulen = ntohs(uh->len);
```

We check to ensure that the skb is sufficiently long to contain a complete UDP header. Pskb_trim checks the packet length in the process of setting tail to point to the end of the UDP header.

```
    if (ulen > len || ulen < sizeof(*uh))
        goto short_packet;
    if (pskb_trim(skb, ulen))
        goto short_packet;
```

The UDP checksum is begun. The ip_summed field in skb is set depending on whether it is necessary to calculate the checksum.

```
    if (udp_checksum_init(skb, uh, ulen, saddr, daddr) < 0)
        goto csum_error;
```

The routing table entry for this incoming packet is checked to see if it was sent to a broadcast or multicast address. If so, we call udp_v4_mcast_deliver to complete the processing.

```
    if(rt->rt_flags & (RTCF_BROADCAST|RTCF_MULTICAST))
        return udp_v4_mcast_deliver(skb, uh, saddr, daddr);
```

Udp_v4_lookup determines if there is an open socket on the UDP port in this packet's header by searching the UDP hash table. If there is an open socket, the packet is passed on to the socket's receive queue and we are done.

```
    if (sk != NULL) {
        int ret = udp_queue_rcv_skb(sk, skb);
        sock_put(sk);
```

If the return value is greater than zero, we must tell the caller to resubmit the input packet.

```
        if (ret > 0)
            return -ret;
        return 0;
    }
    if (!xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb))
        goto drop;
```

If there is no socket for this packet, the checksum calculation is completed. The packet is dropped silently if there is a checksum error. If it is a valid packet but has been sent to a port for which there is no open socket, it is processed as a port- unreachable packet. The UdpNoPorts count is incremented, an ICMP port- unreachable message is sent to the sending machine, and the packet is freed. We are done.

```
    if (udp_checksum_complete(skb))
        goto csum_error;
    UDP_INC_STATS_BH(UdpNoPorts);
    icmp_send(skb, ICMP_DEST_UNREACH, ICMP_PORT_UNREACH, 0);
    kfree_skb(skb);
    return(0);
```

At this point, all that is left is to process the bad packet errors detected earlier.

```
short_packet:
    NETDEBUG(if (net_ratelimit())
             printk(KERN_DEBUG
```

```
            "UDP: short packet: %u.%u.%u.%u:%u %d/%d to %u.%u.%u.%u:%u\ n",
                NIPQUAD(saddr),
                ntohs(uh->source),
                ulen,
                len,
                NIPQUAD(daddr),
                ntohs(uh->dest)));
no_header:
    UDP_INC_STATS_BH(UdpInErrors);
    kfree_skb(skb);
    return(0);
csum_error:
```

Even though the packet is discarded silently, we still increment the UDP error statistics.

```
    NETDEBUG(if (net_ratelimit())
                printk(KERN_DEBUG
                       "UDP: bad checksum. From %d.%d.%d.%d:%d to
                        %d.%d.%d.%d:%d ulen %d\ n",
                       NIPQUAD(saddr),
                       ntohs(uh->source),
                       NIPQUAD(daddr),
                       ntohs(uh->dest),
                       ulen));
drop:
    UDP_INC_STATS_BH(UdpInErrors);
    kfree_skb(skb);
    return(0);
}
```

10.3.2 Receiving Multicast and Broadcast Packets in UDP

Multicast and broadcast packets are sent to multiple destinations. In fact, there may be multiple destinations in the same machine. When UDP receives a multicast or broadcast packet, it checks to see if there are multiple open sockets that should receive the packet. When the routing table entry for the incoming packet has the multicast or broadcast flags set, the UDP receive function, udp_rcv, calls the UDP multicast receive function, udp_v4_mcast_deliver in file *linux/net/ipv4/udp.,c* to distribute the incoming packet to all valid listening sockets.

```
static int udp_v4_mcast_deliver(struct sk_buff *skb, struct udphdr *uh,
                                u32 saddr, u32 daddr)
{
    struct sock *sk;
    int dif;
```

First, we lock the UDP hash table. Then, we select the first socket in the hash table that is open on a port matching the destination port in the header of the incoming packet.

```
    read_lock(&udp_hash_lock);
    sk = sk_head(&udp_hash[ntohs(uh->dest) & (UDP_HTABLE_SIZE - 1)]);
    dif = skb->dev->ifindex;
    sk = udp_v4_mcast_next(sk, uh->dest, daddr, uh->source, saddr, dif);
    if (sk) {
        struct sock *sknext = NULL;
```

We loop through the hash table checking each potential matching entry. When an appropriate listening socket is found, the socket buffer, skb, is cloned and the new buffer is put on the receive queue of the listening socket by calling udp_queue_rcv_skb, which is the UDP backlog receive function discussed in Section 10.2.4. If there is no match, the packet is silently discarded. There is no statistics counter to increment for discarded multicast and broadcast received packets.

```
        do {
            struct sk_buff *skb1 = skb;
            sknext = udp_v4_mcast_next(sk->next, uh->dest, daddr,
                                       uh->source, saddr, dif);
            if(sknext)
                skb1 = skb_clone(skb, GFP_ATOMIC);
            if(skb1)
                int ret = udp_queue_rcv_skb(sk, skb1);
            if (ret > 0)
```

The comments say that we should be reprocessing packets instead of dropping them here. However, as of this kernel revision, we are not.

```
                kfree_skb(skb1);
            sk = sknext;
        }  while(sknext);
    }  else
        kfree_skb(skb);
    read_unlock(&udp_hash_lock);
    return 0;
}
```

10.3.3 UDP Hash Table

As we know from earlier chapters, an application opens a socket of type SOCK_DGRAM to receive UDP packets. This type of socket may listen on a variety of address and port combinations. For example, it may want to receive all packets sent from any address but with a particular destination port, or it may want to receive packets sent to multicast or broadcast addresses. A packet arriving in the udp_rcv function may be passed on to multiple listening sockets. It is not sufficient for UDP to determine which socket or sockets should receive the packet solely by matching the destination port. To determine which socket should get an incoming packet, Linux uses a UDP hash table, udp_hash, defined in the file *udp.c*.

```
struct sock *udp_hash[UDP_HTABLE_SIZE];
```

The hash table can contain up to 128 slots, and each location in the hash table points to a list of sock structures. The hash value used for the table index is calculated from the lowest 7 bits of the UDP port number.

In *linux/net/ipv4/udp.c*, Udp_v4_lookup is a utility function that looks up a socket in the hash table based on matching source and destination ports, source and destination addresses, and network interface. It locks the hash table and calls udp_v4_lookup_longway to do the real work.

```
__inline__ struct sock *udp_v4_lookup(u32 saddr, u16 sport, u32 daddr, u16
                                      dport, int dif)
```

```
{
    struct sock *sk;
    read_lock(&udp_hash_lock);
    sk = udp_v4_lookup_longway(saddr, sport, daddr, dport, dif);
    if (sk)
        sock_hold(sk);
    read_unlock(&udp_hash_lock);
    return sk;
}
```

The next function, Udp_v4_lookup_longway, is called from udp_v4_lookup. It is declared in the file *linux/net/ipv4/udp.c*. It tries to find the socket as best it can. Minimally, it matches the destination port number. Then, it tries to refine the choice further by matching the source address, destination address, and the incoming network interface.

```
struct sock *udp_v4_lookup_longway(u32 saddr, u16 sport, u32 daddr, u16
                                   dport, int dif)
{
    struct sock *sk, *result = NULL;
    struct hlist_node *node;
    unsigned short hnum = ntohs(dport);
    int badness = -1;

    sk_for_each(sk, node, &udp_hash[hnum & (UDP_HTABLE_SIZE - 1)]) {
```

Our idea here is to find the best match of the incoming packet with an open socket. First, we check to make sure that it is not an IPv6-only socket. As discussed in Chapter 5, "Linux Sockets," IPv6 sockets may generally be used for IPv4. In this section of code, we find the first matched socket, and this is the one that gets the packet.

```
        struct inet_opt *inet = inet_sk(sk);
        if (inet->num == hnum && !ipv6_only_sock(sk)) {
```

First, we check that the address family matches.

```
            int score = (sk->sk_family == PF_INET ? 1 : 0);
            if (inet->rcv_saddr) {
                if (inet->rcv_saddr != daddr)
                    continue;
                score+=2;
            }
```

Now we check for a matching destination address.

```
            if (inet->daddr) {
                if (inet->daddr != saddr)
                    continue;
                score+=2;
            }
```

We check for a matching port.

```
            if (inet->dport) {
```

```
                if (inet->dport != sport)
                    continue;
                score+=2;
            }
```

Finally, we check to see if the input network interface matches (but only if it is bound).

```
            if (sk->sk_bound_dev_if) {
                if (sk->sk_bound_dev_if != dif)
                    continue;
                score+=2;
            }
            if(score == 9) {
                result = sk;
                break;
            }  else if(score > badness) {
                result = sk;
                badness = score;
            }
        }
    }
    return result;
}
```

10.3.4 UDP Backlog Receive

The function, udp_queue_rcv_skb, in the file *linux/net/ipv4/udp.c* is the backlog receive function for UDP, called from the "bottom half" when the socket is held by the user and can't accept any more data. This function is initialized at compile time into the backlog_rcv field of the UDP proto structure, udp_prot. As we saw in Section 10.3.1, udp_queue_rcv_skb is called by udp_rcv to complete input packet processing, and its purpose is mainly to place incoming packets on the socket's receive queue and increment the UDP input statistics.

```
static int udp_queue_rcv_skb(struct sock * sk, struct sk_buff *skb)
{
    struct udp_opt *up = udp_sk(sk);
    if (!xfrm4_policy_check(sk, XFRM_POLICY_IN, skb)) {
        kfree_skb(skb);
        return -1;
    }
```

First, we check to see if this is an encapsulated socket for IPSec.

```
    if (up->encap_type) {
```

If we have an encapsulated socket, the incoming packet must be encapsulated. If it is, we transform the input packet. If not, we assume it is an ordinary UDP packet and we fall through.

```
        int ret;
        ret = udp_encap_rcv(sk, skb);
        if (ret == 0) {
            kfree_skb(skb);
            return 0;
        }
```

```
        if (ret < 0) {
```

Here, we process the ESP packet.

```
            ret = xfrm4_rcv_encap(skb, up->encap_type);
            UDP_INC_STATS_BH(UdpInDatagrams);
            return -ret;
        }
```

If we fall through, the packet must be an ordinary UDP packet.

```
    }
```

Now, we finish calculating the checksum.

```
    if (sk->sk_filter && skb->ip_summed != CHECKSUM_UNNECESSARY) {
        if (__udp_checksum_complete(skb)) {
            UDP_INC_STATS_BH(UdpInErrors);
            kfree_skb(skb);
            return -1;
        }
        skb->ip_summed = CHECKSUM_UNNECESSARY;
    }
```

Next, we call the socket-level receive function, sock_queue_rcv_skb, covered in Chapter 5, which places the packet on the socket's receive queue and wakes up the socket so the application can read the data. If there is insufficient room on the receive queue, the error statistics are incremented and the packet is silently discarded.

```
    if (sock_queue_rcv_skb(sk,skb)<0) {
```

If the socket returned an error, we increment the statistics and get out.

```
        UDP_INC_STATS_BH(UdpInErrors);
        IP_INC_STATS_BH(IpInDiscards);
        ip_statistics[smp_processor_id()*2].IpInDelivers–;
        kfree_skb(skb);
        return -1;
    }
```

Once we increment the counter, we are done. At this point, the socket-level processing will allow the user-level read to complete.

```
    UDP_INC_STATS_BH(UdpInDatagrams);
    return 0;
}
```

10.4 UDP Socket-Level Receive

In Chapter 8, we saw how all the socket send calls are converged at the socket layer into one function at the transport layer. On the send side, all the socket-level send functions converge on

one function. For UDP, this function is udp_sendmsg. As is the case with the send side, when the application calls any of the read functions on an open socket, the socket layer calls the function pointed to by the rcvmsg field in the prot structure. At compile time, the rcvmsg field is initialized to udp_rcvmsg. Udp_rcvmsg, in the file *linux/net/ipv4/udp.c*, is the receiving function that is executed for all SOCK_DGRAM type sockets. See Chapter 8 for a description of the protocol block structure for UDP.

```
int udp_recvmsg(struct sock *sk, struct msghdr *msg, int len,
                int noblock, int flags, int *addr_len)
{
    struct inet_opt *inet = inet_sk(sk);
    struct sockaddr_in *sin = (struct sockaddr_in *)msg->msg_name;
    struct sk_buff *skb;
    int copied, err;
```

Set the application's address length argument. Check to see if there are any messages in the error queue of the socket, sk, and if so, process them by calling ip_recv_error and going out.

```
    if (addr_len)
        *addr_len=sizeof(*sin);
    if (flags & MSG_ERRQUEUE)
        return ip_recv_error(sk, msg, len);
```

De-queue packets from the socket sk's receive queue by calling the generic datagram receive queue function, skb_recv_datagram, and if it returns without a packet, get out.

```
    skb = skb_recv_datagram(sk, flags, noblock, &err);
    if (!skb)
        goto out;
```

Check to see if the user is asking for more data than the payload in the packet, skb.

```
    copied = skb->len - sizeof(struct udphdr);
    if (copied > len) {
        copied = len;
        msg->msg_flags |= MSG_TRUNC;
    }
```

The ip_summed field in the socket buffer determines if checksums are required. If ip_summed is not equal to CHECKSUM_UNNECESSARY, checksum calculation must be done, and data is copied from kernel space to user space while calculating a checksum. The copying and checksum calculation is done by skb_copy_and_csum_datagram_iovec. However, if a checksum is not required, skb_copy_datagram_iovec is the function called to do the copying. It is also possible that a partial checksum was calculated because the data spanned more than one buffer, and in this case, __udp_checksum_complete finishes the checksum calculation. A checksum is the amount that makes an integer sum always add up to zero. Checksums can be done partially and finished later.

```
    if (skb->ip_summed==CHECKSUM_UNNECESSARY) {
        err = skb_copy_datagram_iovec(skb, sizeof(struct udphdr),
                                      msg->msg_iov, copied);
```

```
    }  else if (msg->msg_flags&MSG_TRUNC) {
        if (__udp_checksum_complete(skb))
            goto csum_copy_err;
        err = skb_copy_datagram_iovec(skb, sizeof(struct udphdr),
                                      msg->msg_iov,
                                      copied);
    }  else {
        err = skb_copy_and_csum_datagram_iovec(skb, sizeof(struct udphdr),
                                               msg->msg_iov);

        if (err == -EINVAL)
            goto csum_copy_err;
    }
    if (err)
        goto out_free;
```

Next, the incoming packet is timestamped. If the user supplied a valid buffer, sin, to receive the packet's source address and port, the information is copied from the packet header.

```
    sock_recv_timestamp(msg, sk, skb);
    if (sin)
    {
        sin->sin_family = AF_INET;
        sin->sin_port = skb->h.uh->source;
        sin->sin_addr.s_addr = skb->nh.iph->saddr;
        memset(sin->sin_zero, 0, sizeof(sin->sin_zero));
    }
```

Before leaving, udp_recv checks the control message flags field, cmsg_flags, to see if any IP socket options are set. For example, certain socket options such as IP_TOS require parts of the IP header to be copied into user space. If there are any flags set, ip_cmsg_recv retrieves the associated option values.

```
    if (sk->protinfo.af_inet.cmsg_flags)
        ip_cmsg_recv(msg, skb);
    err = copied;
```

There are three error exits at the end of udp_rcv. The most interesting is csum_copy_err.

```
out_free:
    skb_free_datagram(sk, skb);
out:
    return err;
csum_copy_err:
    UDP_INC_STATS_BH(UdpInErrors);
```

When the flags argument is set to MSG_PEEK, it means that the caller wants to "peek" at the incoming message without removing it from the receive queue. However, we know the packet has a bad checksum, so we must remove it from the receive queue before deleting it.

```
    if (flags&MSG_PEEK) {
        int clear = 0;
        spin_lock_irq(&sk->receive_queue.lock);
        if (skb == skb_peek(&sk->receive_queue)) {
```

```
            __skb_unlink(skb, &sk->receive_queue);
            clear = 1;
        }
        spin_unlock_irq(&sk->receive_queue.lock);
        if (clear)
            kfree_skb(skb);
    }
    skb_free_datagram(sk, skb);
    return -EAGAIN;
}
```

10.5 Receiving Data in TCP

In Section 10.3 we showed how the IP protocol dispatches the UDP receive function based on the protocol field of the IP header. This mechanism was covered in detail in Chapter 6. As with UDP, TCP also initializes an instance of the inet_protocol structure. This structure allows the network layer, IP, to dispatch transport layer handler functions based on the value in the protocol field of the IP header without needing to know anything about the internal structure of each transport layer protocol. It is helpful for a good understanding of TCP to be able to visualize the state management. Figure 10.1 shows the TCP receive-side state machine. See Chapter 8, Figure 8.1 for the send-side TCP state diagram.

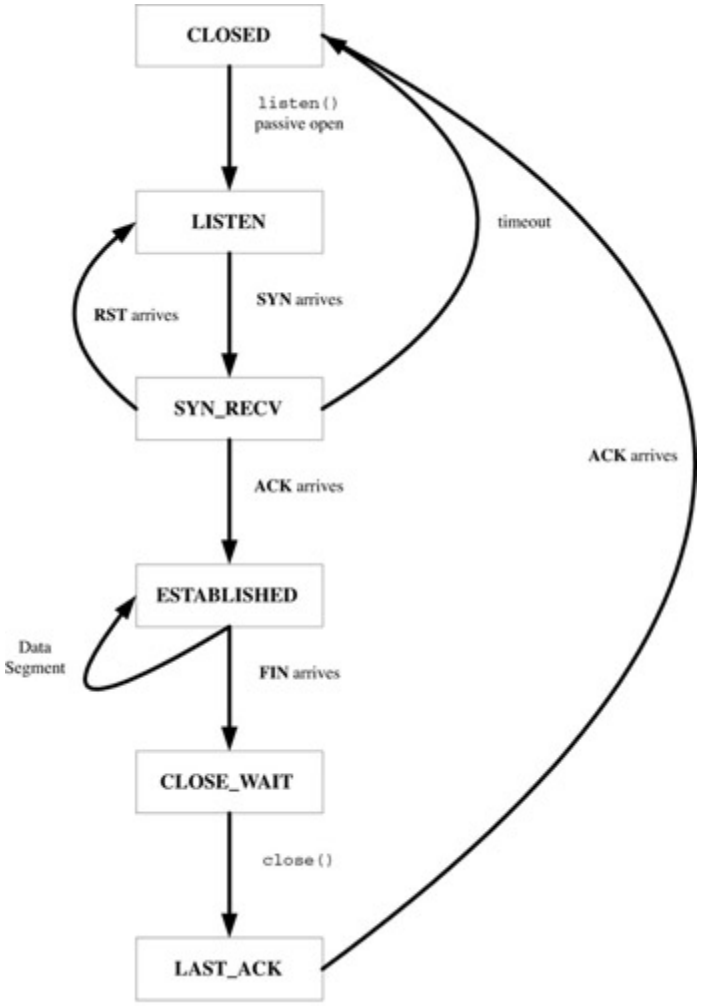

Figure 10.1: TCP receive-side state diagram.

Some key flag definitions are used primarily on the receive side of TCP (see Table 10.1). These defines, found in file *linux/net/ipv4/tcp_input.c*, are primarily used to govern the state processing during the receive side of the TCP connection. The flags are shown in Table 10.1. They are also used as part of the implementation of the slow start and congestion avoidance, selective acknowledgment, and fast acknowledgment, but should not be confused with the flags in the TCP control buffer, which are discussed in Chapter 8.

Table 10.1: Receive State Flags

Flag	**Value**	**Description**
FLAG_DATA	0x01	Incoming frame-contained data.
FLAG_WIN_UPDATE	0x02	Incoming ACK was a window update.
FLAG_DATA_ACKED	0x04	This ACK acknowledged new data.
FLAG_RETRANS_DATA_ACKED	0x08	This ACK acknowledged new data, some of which was retransmitted.
FLAG_SYN_ACKED	0x10	This ACK acknowledged SYN.
FLAG_DATA_SACKED	0x20	New SACK.

Table 10.1: Receive State Flags

Flag	Value	Description
FLAG_ECE	0x40	The ECE flag is set in this ACK.
FLAG_DATA_LOST	0x80	SACK detected data loss.
FLAG_SLOWPATH	0x100	Do not skip RFC checks for window update.
FLAG_ACKED		(FLAG_DATA_ACKED|FLAG_SYN_ACKED)
FLAG_NOT_DUP		(FLAG_DATA|FLAG_WIN_UPDATE|FLAG_ACKED)
FLAG_CA_ALERT		(FLAG_DATA_SACKED|FLAG_ECE)
FLAG_FORWARD_PROGRESS		(FLAG_ACKED|FLAG_DATA_SACKED)

10.5.1 TCP Receive Handler Function, tcp_v4_rcv

In this section, we will examine the TCP input segment handling and the registered handler function for the TCP protocol in the AF_INET protocol family. Figure 10.2 shows the TCP receive packet flow.

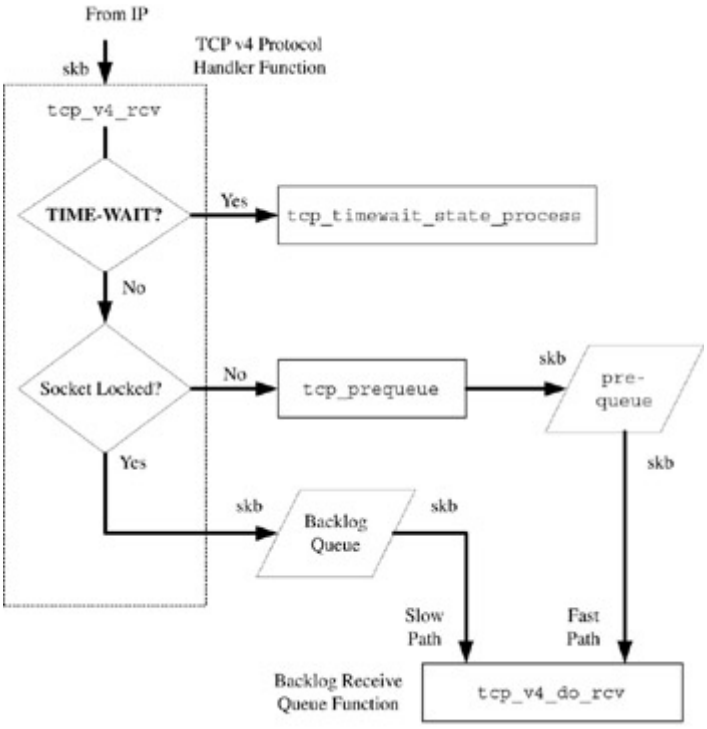

Figure 10.2: TCP receive packet flow.

As is the case with all other member protocols in the AF_INET family, TCP associates a handler function with the protocol field value, IPPPROTO_TCP, (the value six) by initializing an instance of the inet_protocol structure. This process is described in Chapter 6. The handler field is set to the function tcp_v4_rcv. Therefore, tcp_v4_rcv, defined in file *linux/net/ipv4/tcp_ipv4.c*, is called from IPv4 when the protocol type in the IP header contains the protocol number for TCP.

```
int tcp_v4_rcv(struct sk_buff *skb)
{
    struct tcphdr *th;
    struct sock *sk;
    int ret;
    if (skb->pkt_type!=PACKET_HOST)
        goto discard_it;
```

TCP counters are incremented before TCP checksums are validated. The next section of code is validating the TCP, checking that it is complete and that the header length field is at least as big as the TCP header without TCP options. As is the case with UDP, we use the socket buffer utility function pskb_may_pull to ensure that the socket buffer, skb, contains a complete header.

```
    TCP_INC_STATS_BH(TcpInSegs);
    if (!pskb_may_pull(skb, sizeof(struct tcphdr)))
        goto discard_it;
```

Th is set to point to the TCP header in the SKB. The doff field is the 4-bit header length.

```
    th = skb->h.th;
    if (th->doff < sizeof(struct tcphdr)/4)
        goto bad_packet;
    if (!pskb_may_pull(skb, th->doff*4))
        goto discard_it;
```

Here, the checksum is initialized. The TCP header and the IP fake header is used to initialize the checksum. The rest of the checksum calculation is put off until later.

```
    if ((skb->ip_summed != CHECKSUM_UNNECESSARY &&
         tcp_v4_checksum_init(skb) < 0))
        goto bad_packet;
```

A few fields are extracted form the TCP header and updated in the TCP control buffer part in the skb. TCP will want quick access to these values in the TCP packet header to do header prediction, which chooses incoming packets fast path processing. The fields used for header prediction include the TCP sequence number, seq in the control buffer, and the TCP acknowledgment number, ack_seq, in the control buffer. The end_seq is the position of the last byte in the incoming segment. At this point, it is set to the received sequence number, seq, plus the data length in the segment, plus one if this is a SYN or FIN packet. Two other fields in the control buffer, when and sacked, are set to zero. When is used for RTT calculation, and sacked is for selective acknowledgment. The macro TCP_SKB_CB is used to get the pointer to the TCP control buffer from the socket buffer. The control buffer structure is covered in Chapter 8.

```
    th = skb->h.th;
    TCP_SKB_CB(skb)->seq = ntohl(th->seq);
    TCP_SKB_CB(skb)->end_seq = (TCP_SKB_CB(skb)->seq + th->syn + th->fin +
                                skb->len - th->doff*4);
    TCP_SKB_CB(skb)->ack_seq = ntohl(th->ack_seq);
    TCP_SKB_CB(skb)->when = 0;
    TCP_SKB_CB(skb)->flags = skb->nh.iph->tos;
    TCP_SKB_CB(skb)->sacked = 0;
```

Now we try to find an open socket for this incoming segment if there is one. As is the case with most functions in Linux, TCP/IP, the double underscore "__" before tcp_v4_lookup means that the caller must acquire the lock before calling the function. Tcp_v4_lookup attempts to find the socket based on the incoming network interface, the source and destination IP addresses, and the source TCP port. If we have a socket, sk is set to point to the sock structure for the open socket and we continue to process the incoming segment.

```
    sk = __tcp_v4_lookup(skb->nh.iph->saddr, th->source,
                         skb->nh.iph->daddr,
                         ntohs(th->dest),tcp_v4_iif(skb));
    if (!sk)
        goto no_tcp_socket;
```

We process the incoming packet. If IP security is installed, we do the security transformation. Once we have the socket, sk, we can check the state of the connection. If the connection is in the TIME-WAIT state, we must handle any incoming segments in a special way. Once we are in TIME-WAIT, delayed segments must be discarded and an incoming TCP packet may contain a delayed segment.

```
process:
    if (sk->sk_state == TCP_TIME_WAIT)
        goto do_time_wait;
```

Check for the IPSec security transformation.

```
    if (!xfrm4_policy_check(sk, XFRM_POLICY_IN, skb))
        goto discard_and_relse;

    if (sk_filter(sk, skb, 0))
        goto discard_and_relse;
    skb->dev = NULL;
    bh_lock_sock(sk);
    ret = 0;
```

If the socket is locked by the top-half process, it can't accept any more segments, so we must put the incoming segment on the backlog queue by calling sk_add_backlog. If the socket is not locked, we try to put the segments on the prequeue. The prequeue is in the user copy structure, ucopy. Ucopy is part of the TCP options structure, discussed in Chapter 8, Section 8.7.2. Once segments are put on the prequeue, they are processed in the application task's context rather than in the kernel context. This improves the efficiency of TCP by minimizing context switches between kernel and user. If tcp_prequeue returns zero, it means that there was no current user task associated with the socket, so tcp_v4_do_rcv is called to continue with normal "slow path" receive processing. Tcp_prequeue is covered in more detail later in this chapter.

```
    if (!sock_owned_by_user(sk)) {
        if (!tcp_prequeue(sk, skb))
            ret = tcp_v4_do_rcv(sk, skb);
    }  else
        sk_add_backlog(sk, skb);
```

Now we can unlock the socket by calling bh_unlock_sock instead of unlock_sock, because in this function, we are executing in the "bottom half" context. Sock_put decrements the socket reference count indicating that the sock has been processed.

```
    bh_unlock_sock(sk);
    sock_put(sk);
    return ret;
```

The label no_tcp_socket is where we end up if there is no current open TCP socket. We still must complete the checksum to see if the packet is bad. If the packet had a bad checksum, we increment the error counter. If the packet was good, we are here because the segment was sent to a socket that is not open, so we send out a reset request to bring down the connection. Next, we discard the packet and we are free to go.

```
no_tcp_socket:
    if (!xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb))
        goto discard_it;
    if (skb->len < (th->doff<<2) || tcp_checksum_complete(skb)) {
bad_packet:
        TCP_INC_STATS_BH(TcpInErrs);
    }  else {
        tcp_v4_send_reset(skb);
    }
discard_it:
    kfree_skb(skb);
    return 0;
```

We arrived here because we need to discard the packet. We decrement the reference count and free the packet.

```
discard_and_relse:
    sock_put(sk);
    goto discard_it;
```

We jumped here, do_time_wait, because the socket, sk, is in the TIME_WAIT state. The TIME_WAIT state requires a little more discussion because arriving packets require special attention, so it will be covered in Section 10.4.2.

```
do_time_wait:
    if (!xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb))
        goto discard_and_relse;
```

Next, we check to see if the header length is too short and we try to complete the checksum. If these tests fail, we get rid of the packet and go out. The function tcp_checksum_complete should return a zero if it is successful.

```
    if (skb->len < (th->doff<<2) || tcp_checksum_complete(skb)) {
        TCP_INC_STATS_BH(TcpInErrs);
        goto discard_and_relse;
    }
```

Tcp_timewait_state_process looks at the arriving packet, skb, to see whether it is a **SYN**, **FIN**, or a data segment and determines what to do. The states returned by tcp_timewait_state_process are shown in Table 10.2.

```
    switch(tcp_timewait_state_process((struct tcp_tw_bucket *)sk,
                                      skb, th, skb->len)) {
```

Table 10.2: TCP TIME_WAIT Status Values

Symbol	Value	Meaning
TCP_TW_SUCCESS	0	Delayed segment or duplicate ACK arrived. Discard packet.
TCP_TW_RST	1	Received FIN. Send a RST to peer.
TCP_TW_ACK	2	Received last ACK. Send ACK to peer.
TCP_TW_SYN	3	Received SYN. Try to re-open connection.

According to RFC 1122, an arriving **SYN** can "wake up" and re-establish a connection in the **TIME-WAIT** state. In this case, tcp_v4_lookup_listener is called to spawn a listener and establish a new connection.

```
    case TCP_TW_SYN:
    {
        struct sock *sk2 = tcp_v4_lookup_listener(skb->nh.iph->daddr,
                                                  ntohs(th->dest),
                                                  tcp_v4_iif(skb));
        if (sk2) {
            tcp_tw_deschedule((struct tcp_tw_bucket *)sk);
            tcp_timewait_kill((struct tcp_tw_bucket *)sk);
            tcp_tw_put((struct tcp_tw_bucket *)sk);
            sk = sk2;
            goto process;
        }
    }
```

We received the last **ACK** from the peer, so we must send the final **ACK** to close the connection.

```
    case TCP_TW_ACK:
        tcp_v4_timewait_ack(sk, skb);
        break;
```

A **FIN** was received, so we reset the connection by sending an **RST**, done at the no_tcp_socket label.

```
    case TCP_TW_RST:
        goto no_tcp_socket;
```

A duplicate **ACK** or delayed segment was received, so we discard it.

```
    case TCP_TW_SUCCESS:;
    }
    goto discard_it;
}
```

10.5.2 The TCP Fast Path, Prequeue Processing

Linux TCP has two paths for input packet processing, a "slow" path and a "fast" path. The slow path is normal input processing. As discussed in Chapter 5, every socket has a two queues, a receive queue and a backlog queue, which is used if the receive queue is full or the socket is busy. Along the slow path, packets received by TCP are placed on the socket's receive queue only after the packet is determined to be a valid data segment containing in-order segment data. This is a large amount of processing and it is all done in the context of the "bottom half" of Linux, in the context of the network interface receive functions. Once the packets are on the queue, the socket is woken up and the scheduler executes the user-level task, which reads the packets from the queue. As part of slow path processing, when the receive queue is full or the user-level task has the socket locked, the packets are placed on the backlog queue.

In addition to the two queues used in the slow path, Linux TCP has a third queue called the prequeue. This third queue is used for fast path processing. Because of the high volume of data that TCP is expected to handle, Linux includes the fast path as one of the various speedups for optimized performance. As we saw in Chapter 6, one of the important performance enhancing features of the Linux TCP/IP is TCP header prediction. The header prediction algorithm determines if a received packet is likely to be an in-order segment containing data received while the socket is in the **ESTABLISHED** state [JACOB93]. If these conditions are met, the packet is elected for processing by the fast path. In general, the Van Jacobsen algorithm assumes that at least half of all packets arriving while the connection is established will consist of data segments rather than ACK packets. By using header prediction, TCP's low-level receive function tries to determine which of these packets meet the criteria, and those packets are immediately placed on the prequeue. When the user task is woken up, TCP processing of the packets on the prequeue is done by the user-level task bypassing many of the processing steps during the slow path. Fast path processing occurs before the normal processing of any packets on the regular receive queue.

There are three functions and one data structure used with the TCP prequeue. The ucopy structure contains the queue itself. The Tcp_prequeue_init function, in the file *linux/net/ipv4/tcp_ipv4.c*, initializes the prequeue, and the inline function tcp_prequeue in file *linux/include/net/tcp.h* puts packets on the queue. Another function, tcp_prequeue_process in file *linux/net/ipv4/tcp.c*, removes the data while running in the application task's context when the socket receive function is called.

The prequeue is within the ucopy structure also defined in *file linux/include/linux/tcp.h*, which is in the TCP options part of the sock structure. See Chapter 8 for more details about the basic sock structure. The field, prequeue, contains the list of socket buffers waiting for processing. Task is the user-level task to receive the data. The iov field points to the user's receive data array, and memory contains the sum of the actual data lengths of all of the socket buffers on the prequeue. Len is the number of buffers on the prequeue.

```
struct {
    struct sk_buff_head prequeue;
    struct task_struct  *task;
    struct iovec        *iov;
    int                 memory;
    int                 len;
    }  ucopy;
```

Tcp_prequeue_init, defined in file *linux/linux/net/tcp.h*, initializes the elements of the ucopy structure including the list of socket buffers in the prequeue. This initialization occurs whenever a socket of type SOCK_STREAM is open for the AF_INET address family as part of the initialization of socket state information by the function tcp_v4_init_sock.

```
static __inline__ void tcp_prequeue_init(struct tcp_opt *tp)
{
    tp->ucopy.task = NULL;
    tp->ucopy.len = 0;
    tp->ucopy.memory = 0;
    skb_queue_head_init(&tp->ucopy.prequeue);
}
```

Tcp_prequeue, the function that puts socket buffers on the prequeue, is also defined in file *linux/include/net/tcp.h*. It queues up buffers only if there is a user task currently waiting on the socket. This is indicated by a non-NULL value in the task field of the ucopy structure. When the socket is woken up and a read is issued on the socket by the application, tcp_prequeue immediately processes the socket's prequeue before "officially" calling the socket's receive function through the system call interface.

```
static __inline__ int tcp_prequeue(struct sock *sk, struct sk_buff *skb)
{
```

Tp is set to the TCP options structure, which is retrieved from the sock structure.

```
    struct tcp_opt *tp = &sk->tp_pinfo.af_tcp;
```

Here we check to see if a user task is currently waiting on this socket. If so, we place the socket buffer, skb, on the end of the queue. In addition, the user can control this behavior via sysctl, so we check here.

```
    if (!sysctl_tcp_low_latency && tp->ucopy.task) {
        __skb_queue_tail(&tp->ucopy.prequeue, skb);
        tp->ucopy.memory += skb->truesize;
```

If queuing the current TCP segment, skb, causes the prequeue to grow larger than the size of the socket's receive buffer, the socket buffers on the prequeue are removed and the socket's backlog receive function, backlog_rcv, is called to put each buffer on the backlog queue until all the buffers on the prequeue are handled. After this, the prequeue is reset to empty by setting the memory field in ucopy to zero.

```
        if (tp->ucopy.memory > sk->rcvbuf) {
            struct sk_buff *skb1;
            if (sk->lock.users)
                out_of_line_bug();
            while ((skb1 = __skb_dequeue(&tp->ucopy.prequeue)) != NULL) {
                sk->backlog_rcv(sk, skb1);
                NET_INC_STATS_BH(TCPPrequeueDropped);
            }
            tp->ucopy.memory = 0;
```

When there is only one skb remaining in the prequeue, the socket is woken up. Recall in Chapter 8, Section 8.8.4 about the delayed acknowledgment timer, we discussed how outgoing **ACKs** are held back waiting to be piggybacked on data segments. Therefore, while processing the prequeue, we reset the delayed acknowledgment timer to wait for a send-side segment to carry the **ACK**.

```
        }  else if (skb_queue_len(&tp->ucopy.prequeue) == 1) {
```

This is where we wake up the socket so the prequeue will be processed.

```
            wake_up_interruptible(sk->sleep);
            if (!tcp_ack_scheduled(tp))
                tcp_reset_xmit_timer(sk, TCP_TIME_DACK, (3*TCP_RTO_MIN)/4);
        }
```

We return a one if we have placed the packet on the prequeue, and we return a zero if the packet was not queued. The zero return indicates to the caller that the prequeue was full and the packets were put on the backlog receive queue.

```
        return 1;
    }
    return 0;
}
```

10.5.3 TCP Backlog Queue Processing

The function tcp_v4_do_rcv is the backlog receive function for TCP. It is set in the backlog_rcv field of the proto structure in file *linux/net/ipv4/tcp_ipv4.c*. Tcp_v4_do_rcv is called when the socket can't accept any more incoming data on its receive queue.

```
int tcp_v4_do_rcv(struct sock *sk, struct sk_buff *skb)
{
```

First, we check to see if we are in the **ESTABLISHED** state. If so, the skb is a likely candidate for fast path processing. Tcp_rcv_established is called to do the processing of the input packet in the **ESTABLISHED** state. It does the header prediction to see if the skb can be processed in the fast path. Tcp_rcv_established returns a one if we must send a reset to the other side of the connection, and on successful processing of the packet, it returns a zero.

```
    if (sk->state == TCP_ESTABLISHED) {
        TCP_CHECK_TIMER(sk);
        if (tcp_rcv_established(sk, skb, skb->h.th, skb->len))
            goto reset;
        TCP_CHECK_TIMER(sk);
        return 0;
    }
```

Here we check to see if the packet does not have a complete header ,and we complete the calculation of the checksum. Tcp_checksum_complete returns a zero if the checksum is OK or it is already done, and a nonzero value if the checksum failed.

```
    if (skb->len < (skb->h.th->doff<<2) || tcp_checksum_complete(skb))
```

```
        goto csum_err;
```

If we are in the listening state, we check to see if the incoming packet, skb, is a SYN packet , which would be a connection request. If we receive a valid **SYN**, we must transition to the receive state. This processing is done by tcp_v4_hnd_req, which validates the connection request and returns a sock structure, nsk or NULL. The returned sock will be in the **ESTABLISHED** state.

```
    if (sk->state == TCP_LISTEN) {
        struct sock *nsk = tcp_v4_hnd_req(sk, skb);
        if (!nsk)
            goto discard;
```

Tcp_child_process continues with receive state processing on the child socket, nsk. It returns a zero if the skb was processed successfully, and returns a nonzero value if we must send a reset to the peer, generally because we received a reset during the brief amount of time the child socket was in the **SYN** received state. After all this has been successfully completed, the new child socket, nsk, will be in the **ESTABLISHED** state, ready to transfer data and we are done processing the incoming packet.

```
        if (nsk != sk) {
            if (tcp_child_process(sk, nsk, skb))
                goto reset;
            return 0;
    }
    }
```

If we are not in the **LISTEN** state, we proceed with normal state processing by calling tcp_rcv_state_process, which returns a zero if the skb was successfully processed and returns a one if we must send a reset request to the peer.

```
    TCP_CHECK_TIMER(sk);
    if (tcp_rcv_state_process(sk, skb, skb->h.th, skb->len))
        goto reset;
    TCP_CHECK_TIMER(sk);
    return 0;
```

These three exit labels—reset, discard, and csum_err—are for sending a reset to the peer, discarding the skb, and incrementing the TCP input error counter, respectively. The error counter is incremented if a bad packet was detected.

```
reset:
    tcp_v4_send_reset(skb);
discard:
    kfree_skb(skb);
    return 0;
csum_err:
    TCP_INC_STATS_BH(TcpInErrs);
    goto discard;
}
```

10.6 TCP Receive State Processing

In Chapter 6, we discussed the 11 states of TCP and presented a general overview of how and where they are implemented in the Linux kernel. We know that as a TCP packet arrives, TCP must detect whether the packet contains a data segment or whether it contains a signal, which is one of the bits in the TCP header, **SYN**, **FIN**, **RST**, or **ACK**. This section, along with Sections 10.7 and 10.8, explains the code that does the TCP receive processing. This section covers all TCP state processing as mandated by RFC 793 except the **ESTABLISHED** and the **TIME-WAIT** states, which are discussed in Sections 10.7 and 10.8, respectively. [RFC793] Tcp_rcv_state_process is in the file *linux/net/ipv4/tcp_input.c*. It is the function that does most of the work processing the TCP state of the open socket, and generally it follows RFC 793 where it specifies how to process incoming segments (see RFC 793, Section 3.9, page 64). The function returns a zero if the segment was successfully processed, and returns a one if the caller must send a reset.

```
int tcp_rcv_state_process(struct sock *sk, struct sk_buff *skb,
                          struct tcphdr *th, unsigned len)
{
    struct tcp_opt *tp = &(sk->tp_pinfo.af_tcp);
    int queued = 0;
```

Saw_tstamp is set later to a nonzero value if the packet, skb, contains the timestamp option.

```
    tp->saw_tstamp = 0;
```

First, we handle the CLOSE, LISTEN, and SYN_SENT states. If the state is CLOSED, we discard the incoming segment.

```
    switch (sk->state) {
    case TCP_CLOSE:
        goto discard;
```

If we are in the **LISTEN** state, this socket is acting as a server and is waiting for a connection. If the incoming segment includes an **ACK**, we must send a reset. If the incoming segment contains a **SYN**, we process the connection request. If our connection is in the **RST** state, we discard the incoming packet.

```
    case TCP_LISTEN:
        if(th->ack)
            return 1;
        if(th->rst)
            goto discard;
```

This function is independent of both IPv4 and IPv6. However, where actions taken are specific to an address family, the function in the af_specific part of the tcp_opt structure is used. Recall that the tcp_opt, which was initialized when the socket, sk, was opened for one of the address families, AF_INET or AF_INET6.

```
        if(th->syn) {
```

```
            if(tp->af_specific->conn_request(sk, skb) < 0)
                return 1;
            goto discard;
        }
```

In the **LISTEN** state, any incoming packets that do not contain a SYN or an ACK are discarded. There is discussion in the comments of this function about whether we should process any data in the incoming segment based on the behavior of TCP over non-IP network protocols; however, the action in the current kernel release is to discard the segment.

```
        goto discard;
```

If the current state of this socket is **SYN_SENT**, we must check for the **ACK** or **SYN** flags in the incoming segment to see if we should advance the state to **ESTABLISHED**. The function tcp_rcv_synsent_state_process does most of the work for the SYN_SENT state as specified in RFC 793, pages 67 through 68.

```
    case TCP_SYN_SENT:
        queued = tcp_rcv_synsent_state_process(sk, skb, th, len);
        if (queued >= 0)
            return queued;
```

The comments contain some discussion about how to handle data in received segments while in the **SYN_SENT** state, and tcp_rcv_synsent_state_process can return a negative one if there is data to process. However, other than checking the **URG** flag, we do nothing with the data. A nonzero return indicates that a reset must be sent.

```
        tcp_urg(sk, skb, th);
        __kfree_skb(skb);
        tcp_data_snd_check(sk);
        return 0;
```

Timestamps are checked here before the receive state processing is completed. This is part of the check for wrapped sequence numbers, called Protection Against Wrapped Sequence Numbers (PAWS). Sequence numbers can wrap when the sequence value reaches the maximum integer value and increments to zero. Under conditions of high-speed networks with a high window scaling factor, it is possible for an “old” retransmitted segment to reappear with the same sequence number as the current wrapped value. To prevent this from happening, timestamps are used. The receiver checks the timestamps for ascending values, and any received segment with an earlier timestamp is thrown out. Refer to Stevens, Section 24.6, for more information on PAWS [STEV94]. Tcp_fast_parse_options updates three fields of the tcp_opt structure with the two values in the timestamp TCP option of the incoming packet. The rcv_tsval field is set to the received timestamp, and the rcv_tsecr gets the value of the echo reply timestamp. In addition, saw_tstamp is set, if the timestamp option was detected in the packet. Tcp_fast_parse_options returns a zero if skb contains a short TCP header with no options at all. Tcp_paws_discard does the PAWS processing by checking the timestamp values and returns a one if it detects a segment that should be discarded. For a complete discussion of PAWS, see Stevens [STEV94] and RFC 1323.

```
    if (tcp_fast_parse_options(skb, th, tp) && tp->saw_tstamp &&
```

```
                                tcp_paws_discard(tp, skb)) {
```

A reset packet is always accepted even if it flunks the PAWS test; otherwise, tcp_send_dupack enters quickack mode and sends a duplicate selective acknowledgment of the offending segment indicating to the sender that the packet, skb, is a duplicate segment.

```
        if (!th->rst) {
            NET_INC_STATS_BH(PAWSEstabRejected);
            tcp_send_dupack(sk, skb);
            goto discard;
        }
    }
```

At this point in the tcp_rcv_state_process function, we process all the states other than the **SYN_SENT**, **LISTEN**, and **CLOSE** states, which were handled earlier in the function. The steps for processing the remaining eight states are specified starting on page 69 of RFC 793, and tcp_rcv_state_process follows these steps closely. Step one is to validate the sequence number. Tcp_sequence sees if the sequence number is within the window. It returns a zero if the sequence number is outside of the current window. If we received a bad sequence number, we negatively acknowledge it by calling tcp_send_dupack and discard the incoming segment. As specified by RFC 793, page 69, if the incoming packet contains the reset flag, RST, we drop the segment and return.

```
    if (!tcp_sequence(tp, TCP_SKB_CB(skb)->seq, TCP_SKB_CB(skb)->end_seq)) {
        if (!th->rst)
            tcp_send_dupack(sk, skb);
        goto discard;
    }
```

Step two is to check to see if the incoming packet is a reset. The LISTEN state was already handled earlier in this function. Tcp_reset will process the received reset request for all the other states. It sets the appropriate errors in the sock structure and the socket is destroyed.

```
    if(th->rst) {
        tcp_reset(sk);
        goto discard;
    }
```

Tcp_replace_ts_recent stores the timestamp in the tcp_opt structure, tp.

```
    tcp_replace_ts_recent(tp, TCP_SKB_CB(skb)->seq);
```

Step three on page 71 in RFC 793 is to check security and precedence and is ignored. Step four in the RFC is to check for a received **SYN** in the incoming packet that is inside the current window. A received **SYN** in the window is an error condition so the connection is reset.

```
    if (th->syn && !before(TCP_SKB_CB(skb)->seq, tp->rcv_nxt)) {
        NET_INC_STATS_BH(TCPAbortOnSyn);
        tcp_reset(sk);
        return 1;
    }
```

Step five is to check for an **ACK** in the received packet, skb. This step is fairly complicated in the code, but essentially, if the acknowledge is acceptable, we proceed to the **ESTABLISHED** state.

```
    if (th->ack) {
```

Tcp_ack processes incoming **ACKs**, and the FLAG_SLOWPATH argument tells tcp_ack to do comprehensive checking as well as update the window. It returns a one if the **ACK** is acceptable.

```
        int acceptable = tcp_ack(sk, skb, FLAG_SLOWPATH);
        switch(sk->sk_state) {
```

If the **ACK** was acceptable and the connection is in the **SYN_RECV** state, we most likely are a server in the act of doing a "passive open." We should move to the **ESTABLISH** state.

```
        case TCP_SYN_RECV:
            if (acceptable) {
                tp->copied_seq = tp->rcv_nxt;
                mb();
                tcp_set_state(sk, TCP_ESTABLISHED);
                sk->sk_state_change(sk);
```

Sk_wake_async wakes up the socket, but when the socket, sk, has been created with an active open, we receive a **SYN** on the client side of the connection when **SYN** packets are crossed. A socket that is being established with a passive open will not be woken up because the socket field of sk is NULL.

```
                if (sk->sk_socket) {
                    sk_wake_async(sk,0,POLL_OUT);
                }
```

Now we update **SND_UNA** as specified in RFC 793, page 72. We also update **SND_WND** with the advertised window shifted by the current scaling factor, snd_wscale, which holds the value in the most recent window scaling option received from the peer. Tcp_init_wl updates snd_wl1 to hold the sequence number in the incoming packet.

```
                tp->snd_una = TCP_SKB_CB(skb)->ack_seq;
                tp->snd_wnd = ntohs(th->window) << tp->snd_wscale;
                tcp_init_wl(tp, TCP_SKB_CB(skb)->ack_seq,
                            TCP_SKB_CB(skb)->seq);
```

Now that we are moving the socket to the ESTABLISHED state, we must do some housekeeping to make the socket ready to receive data segments. First, tcp_ack did not calculate the RTT, so the RTT is determined based on the timestamps if there was a timestamp option in the received ACK packet; the RTT value is kept in the srtt field of tcp_opt structure. The MSS is adjusted to allow for the size of the timestamp option. Tcp_init_metrics initializes some metrics and calculations for the socket. Next, the received MSS value is set to an initial guess based on the received window size, **RCV.WND**. Buffer space in the socket is reserved based on the received MSS and other factors. Last, tcp_fast_path_on calculates the pred_flags field in tcp_opt, which determines if the receive fast path is on, whether header prediction will be used.

```
                if (tp->saw_tstamp && tp->rcv_tsecr && !tp->srtt)
                    tcp_ack_saw_tstamp(tp, 0);
                if (tp->tstamp_ok)
                    tp->advmss -= TCPOLEN_TSTAMP_ALIGNED;
                tp->af_specific->rebuild_header(sk);
                tcp_init_metrics(sk);
```

Prevent spurious congestion window restart on the first data packet.

```
                tp->lsndtime = tcp_time_stamp;
                tcp_initialize_rcv_mss(sk);
                tcp_init_buffer_space(sk);
```

We turn on the fast path since we are going to the ESTABLISHED state.

```
                tcp_fast_path_on(tp);
            }  else {
```

If the incoming acknowledge packet was not acceptable, we return a one, which tells the caller (tcp_v4_do_rcv) to send a reset.

```
                return 1;
            }
            break;
```

If the connection is in the **FIN_WAIT_1** state and we receive an **ACK**, we enter the **FIN_WAIT_2** state.

```
        case TCP_FIN_WAIT1:
            if (tp->snd_una == tp->write_seq) {
                tcp_set_state(sk, TCP_FIN_WAIT2);
```

Setting shutdown to SEND_SHUTDOWN indicates that a shutdown (a packet containing an **RST**) should be sent to the peer later when the socket moves to the **CLOSED** state.

```
                sk->shutdown |= SEND_SHUTDOWN;
                dst_confirm(sk->sk_dst_cache);

                if (!sk->dead)
```

If the socket is not orphaned, we wake it up, which moves it into the **FIN_WAIT_2** state.

```
                    sk->sk_state_change(sk);
                else {
```

Otherwise, we process the Linux TCP socket option, TCP_LINGER2. The value for this option is in the linger2 field of tcp_opt, which controls how long we remain in the **FIN_WAIT_2** state before proceeding to the **CLOSED** state. However, if linger2 is negative, we proceed directly to **CLOSED** without passing through the **FIN_WAIT_2** and the **TIME_WAIT** states.

```
                    int tmo;
                    if (tp->linger2 < 0 ||
```

```
                    (TCP_SKB_CB(skb)->end_seq != TCP_SKB_CB(skb)->seq &&
                    after(TCP_SKB_CB(skb)->end_seq - th->fin,
                          tp->rcv_nxt))) {
                    tcp_done(sk);
                    NET_INC_STATS_BH(TCPAbortOnData);
                    return 1;
                }
```

Tcp_fin_time checks for the keepalive option, SO_LINGER. It calculates the time to wait in the **FIN_WAIT_2** state based on the option value, default settings, and the re-transmit time. The keepalive timer is reset with the calculated timeout value.

```
                tmo = tcp_fin_time(tp);
                if (tmo > TCP_TIMEWAIT_LEN) {
                    tcp_reset_keepalive_timer(sk,
                          tmo - TCP_TIMEWAIT_LEN);
                }  else if (th->fin || sock_owned_by_user(sk)) {
```

If the incoming **ACK** included a **FIN** or the socket is locked, we reset the keepalive timer. The comment in the code states that if we don't do this, we could lose the incoming FIN.

We proceed as if the SO_LINGER option was selected so input state processing for the **FIN** state will resume in the keepalive timer function. Effectively, this should advance the state to **TIME_WAIT** when the keepalive timer expires. See Section 8.8.5 in Chapter 8 for more about the keepalive timer.

```
                    tcp_reset_keepalive_timer(sk, tmo);
                }  else {
                    tcp_time_wait(sk, TCP_FIN_WAIT2, tmo);
                    goto discard;
                }
            }
        }
        break;
```

Now we continue our processing of the incoming **ACK** packet (step five, page 72 in RFC 793) by looking at the **CLOSING** and **LAST_ACK** states. If we are in the **CLOSING** state and we receive an **ACK**, we proceed directly to **TIME_WAIT** provided there is no outstanding data left for the send side of the connection to handle. If we are in the **LAST_ACK** state, we are in the process of doing a passive close, responding to a close call in the peer. An **ACK** received in this state means that we can close the socket, so we call tcp_done.

```
    case TCP_CLOSING:
        if (tp->snd_una == tp->write_seq) {
            tcp_time_wait(sk, TCP_TIME_WAIT, 0);
            goto discard;
        }
        break;
    case TCP_LAST_ACK:
        if (tp->snd_una == tp->write_seq) {
            tcp_update_metrics(sk);
            tcp_done(sk);
            goto discard;
```

```
            }
            break;
        }
    else
        goto discard;
```

At this point, we are done with processing an incoming **ACK**. The sixth step (RFC 793, page 73) is to process an urgent request, so we check the **URG** bit in the incoming segment. This check is done in function tcp_urg, which continues with the processing of the urgent data.

```
    tcp_urg(sk, skb, th);
```

Step seven (RFC 793, page 74) is to process segment text.

```
    switch (sk->state) {
    case TCP_CLOSE_WAIT:
    case TCP_CLOSING:
    case TCP_LAST_ACK:
        if (!before(TCP_SKB_CB(skb)->seq, tp->rcv_nxt))
            break;
    case TCP_FIN_WAIT1:
    case TCP_FIN_WAIT2:
```

RFC 793 specifies that we should queue up the received data received in one of these five states. RFC 1122 specifies that we should send a **RST**, and this is what the BSD operating system does. Starting with version 4.4, Linux does a reset, too.

```
        if (sk->shutdown & RCV_SHUTDOWN) {
            if (TCP_SKB_CB(skb)->end_seq != TCP_SKB_CB(skb)->seq &&
                after(TCP_SKB_CB(skb)->end_seq - th->fin, tp->rcv_nxt)) {
                NET_INC_STATS_BH(TCPAbortOnData);
                tcp_reset(sk);
                return 1;
            }
        }
```

Here we have the normal case where a data segment is received in the **ESTABLISHED** state; tcp_data_queue is called to continue the processing and put the data segment on the socket's input queue.

```
    case TCP_ESTABLISHED:
        tcp_data_queue(sk, skb);
        queued = 1;
        break;
    }
```

The following two functions, tcp_data_snd_check and tcp_ack-snd, determine if a data segment or an **ACK**, respectively, needs to be sent to the peer. The comment here states that "tcp_data could move socket to **TIME_WAIT**," but it is not clear how that can occur.

```
    if (sk->state != TCP_CLOSE) {
        tcp_data_snd_check(sk);
        tcp_ack_snd_check(sk);
```

```
    }
    if (!queued) {
discard:
        __kfree_skb(skb);
    }
    return 0;
}
```

10.7 TCP Processing Data Segments in Established State

Obviously, the purpose of TCP is to transfer data reliably and rapidly. Data is transferred between the peers while the connection is in the **ESTABLISHED** state. Once the socket is in the **ESTABLISHED** state, the function of the connection is to transfer data between the two sides of the connection as fast as the network and the peer will permit. The tcp_v4_do_rcv function covered earlier in the text checks to see if the socket is in the **ESTABLISHED** state while processing incoming packets. If so, the function tcp_rcv_established in file *linux/net/ipv4/tcp_input.c* is called to complete the processing. The primary purpose of this function is to copy the data from the segments to user space as efficiently as possible. Figure 10.3 shows the Linux TCP **ESTABLISHED** state processing.

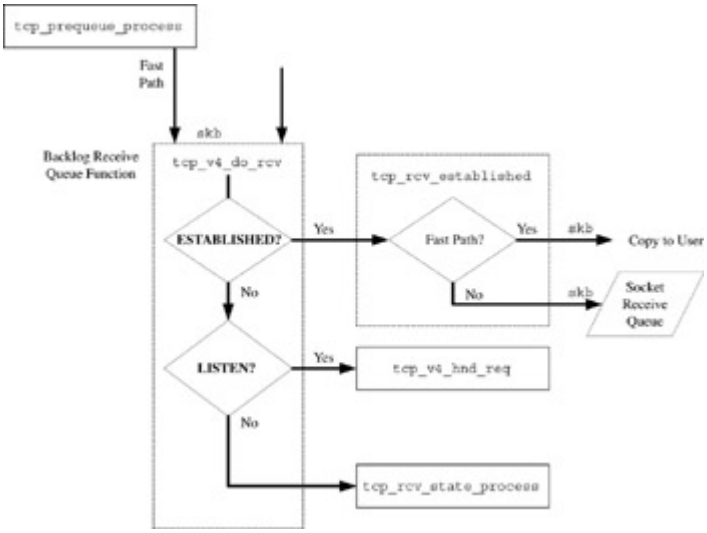

Figure 10.3: TCP established state.

The Linux kernel provides a fast path processing path to speed up TCP data transfer as much as possible during normal conditions where data is being copied through an open socket. It uses a method very similar to Van Jacobson's e-mail about how to do TCP in 30 instructions [JACOB93]. The Linux method differs in that it does the header prediction in advance during the early part of the TCP receive path (see Section 10.5.1). Although somewhat different from Van Jacobson's method, Linux perhaps improves the method in that the completion of fast path processing happens in the application program's task rather than in the "bottom half" of kernel processing. Wherever possible, Linux uses header prediction to select the packets that are most likely to be "normal" data segments for fast path processing. If these packets are in-order data segments, they won't require any processing other than copying the data into the application

program's receive buffer. Therefore, the fast path processing resumes in the context of the application program task while it is executing one of the socket read API functions.

If all incoming TCP packets were put through the fast path, it would no longer be fast. Therefore, the key to the fast path processing is to choose which packets have a high probability of being "normal" data segments that don't require any special time-consuming handling. To accomplish this, we do header prediction to quickly mark these candidate packets early in the receive packet path. Prediction flags are calculated, which are later used to direct packets either to the fast path or the slow path. The header prediction value is calculated as follows:

prediction flags = hlen << 26 ^ ackw ^**SND.WND**,

hlen is the header length

ackw is the BOOLEAN **ACK** << 20

The preceding formula yields a prediction flags value that is equal to bytes 13 through 20 (in network byte order) of the TCP header expressed as a 32-bit word. Thus, the predication flags will be exactly equal to the third 32-bit word in the header of an input segment that consists of a data segment with an **ACK** but no TCP options. The prediction flags are stored in the TCP options structure tp->pred_flags. It is calculated by the inline function, __tcp_fast_path_on, in file *linux/include/net/tcp.h*.

Even though header prediction flags were already calculated, while processing incoming packets in the **ESTABLISHED** state, there are a few checks that cause packets to be redirected to the slow path.

- Our side of the connection announced a zero window. The processing of zero window probes is only handled properly in the slow path. See Chapter 8, Section 8.9 for more information about zero window probes.
- If this side of the conditions receives any out-of-order segments, fast path processing is disabled.
- If urgent data is encountered, fast path processing is disabled until the urgent data is copied to the user.
- If there is no more receive buffer space, the fast path is disabled.
- Any failure of header prediction will divert a particular segment into the slow path.
- The fast path is only supported for unidirectional data transfers, so if we have to send data in the other direction, we default to the slow path processing of incoming segments.
- If there are any options other than a timestamp in the incoming packet, we divert it to the slow path.

Tcp_rcv_established in file *linux/net/ipv4/tcp_input.c* is the function that does the heavy lifting of processing incoming data segments. This function is called only for packets received while the connection is already in the **ESTABLISHED** state. Therefore, it starts out with the assumption that the packets are to be processed in the fast path. Along the way, if it finds that the incoming packet needs a closer look, it is diverted to the slow path.

```
int tcp_rcv_established(struct sock *sk, struct sk_buff *skb,
```

```
                        struct tcphdr *th, unsigned len)
{
    struct tcp_opt *tp = &(sk->tp_pinfo.af_tcp);
    tp->saw_tstamp = 0;
```

Here is where the prediction flags are compared to the incoming segment. We also check to see that the sequence number is in order. The **PSH** flag in the incoming packet is ignored.

```
    if ((tcp_flag_word(th) & TCP_HP_BITS) == tp->pred_flags &&
        TCP_SKB_CB(skb)->seq == tp->rcv_nxt) {
        int tcp_header_len = tp->tcp_header_len;
```

Check to see if there are any other options other than timestamp, and if so, send this packet to the slow path. The TCP options field saw_tstamp indicates that the incoming packet contained a timestamp option.

```
        if (tcp_header_len == sizeof(struct tcphdr) +
                                    TCPOLEN_TSTAMP_ALIGNED) {
            __u32 *ptr = (__u32 *)(th + 1);
```

If the packet contains TCP options, we send it to the slow path.

```
            if (*ptr != ntohl((TCPOPT_NOP << 24) | (TCPOPT_NOP << 16)
                              | (TCPOPT_TIMESTAMP << 8) | TCPOLEN_TIMESTAMP))
                goto slow_path;

            tp->saw_tstamp = 1;
            ++ptr;
            tp->rcv_tsval = ntohl(*ptr);
            ++ptr;
            tp->rcv_tsecr = ntohl(*ptr);
```

Now we do a quick check for PAWS. If the check fails, the packet gets a closer look in the slow path.

```
            if ((s32)(tp->rcv_tsval - tp->ts_recent) < 0)
                goto slow_path;
```

Here we check if the header length is too small and if there is any data in the packet.

```
        if (len <= tcp_header_len) {
```

Here we check for the sending-side fast path. Essentially, we see if we are receiving packets with a valid header but no data, which could indicate that we are doing a one-way data (bulk) transfer in the outgoing direction. Therefore, we acknowledge the packet, free it, and check the send side. We don’t checksum the incoming packet, however, because it has been already been done for header-less packets.

```
            if (len == tcp_header_len) {
```

The predicted packet is in the window.

```
            if (tcp_header_len == (sizeof(struct tcphdr) +
                TCPOLEN_TSTAMP_ALIGNED) && tp->rcv_nxt == tp->rcv_wup)
                tcp_store_ts_recent(tp);
            tcp_ack(sk, skb, 0);
            __kfree_skb(skb);
            tcp_data_snd_check(sk);
            return 0;
```

The header is too small, so throw the packet away.

```
        }  else {
            TCP_INC_STATS_BH(TcpInErrs);
            goto discard;
        }
    }  else {
        int eaten = 0;
```

The global current always points to the currently running task, which is the task at the head of the list of tasks in the TASK_RUNNING state. We check to see if we are running in the context of the application task by seeing if the task structure pointer in ucopy is the same as current. Current was saved in the ucopy structure by tcp_recvmsg, the socket-level receive function when it was called by the application program (through the Linux system call interface, of course).

```
        if (tp->ucopy.task == current &&
            tp->copied_seq == tp->rcv_nxt &&
            len - tcp_header_len <= tp->ucopy.len &&
            sock_owned_by_user(sk)) {
            __set_current_state(TASK_RUNNING);
```

Here is where we actually copy the data. If the data was successfully copied, we update rcv_nxt (the variable, **RCV.NXT** in RFC 793), the next expected receive sequence number. While copying the data, cp_copy_to_iovec also completes the checksum if it wasn't already done by the network interface hardware.

```
            if (!tcp_copy_to_iovec(sk, skb, tcp_header_len)) {
                if (tcp_header_len ==
                     (sizeof(struct tcphdr) +
                      TCPOLEN_TSTAMP_ALIGNED) &&
                     tp->rcv_nxt == tp->rcv_wup)
                    tcp_store_ts_recent(tp);
                __skb_pull(skb, tcp_header_len);
                tp->rcv_nxt = TCP_SKB_CB(skb)->end_seq;
                NET_INC_STATS_BH(TCPHPHitsToUser);
                eaten = 1;
            }
        }
```

We are here either because there isn't a user task context or the copy to user failed. If the copy to user failed, it is probably because of a bad checksum. We complete the checksum if necessary, and if it is bad, we go out. If there is no room in the socket, we complete processing in the slow path.

```
        if (!eaten) {
```

```
            if (tcp_checksum_complete_user(sk, skb))
                goto csum_error;
            if (tcp_header_len ==
                (sizeof(struct tcphdr) + TCPOLEN_TSTAMP_ALIGNED) &&
                tp->rcv_nxt == tp->rcv_wup)
                tcp_store_ts_recent(tp);
            if ((int)skb->truesize > sk->forward_alloc)
                goto step5;
            NET_INC_STATS_BH(TCPHPHits);
```

This is the receiver side of a bulk data transfer. We remove the header portion from the skb and put the data part of the segment on the receive queue.

```
            __skb_pull(skb,tcp_header_len);
            __skb_queue_tail(&sk->sk_receive_queue, skb);
            tcp_set_owner_r(skb, sk);
            tp->rcv_nxt = TCP_SKB_CB(skb)->end_seq;
        }
```

Since we know we received a valid packet, the next few sections of code deal with the sending and receiving acknowledgments. Tcp_event_data_recv updates the delayed acknowledge timeout interval. See Section 8.8.4 in Chapter 8 for more about the delayed acknowledgment system. Tcp_ack handles incoming **ACK**s.

```
        tcp_event_data_recv(sk, tp, skb);
        if (TCP_SKB_CB(skb)->ack_seq != tp->snd_una) {
```

If there is no need to send an **ACK**, we jump the next little section.

```
            tcp_ack(sk, skb, FLAG_DATA);
            tcp_data_snd_check(sk);
            if (!tcp_ack_scheduled(tp))
                goto no_ack;
        }
```

We send an **ACK** if necessary.

```
        if (eaten) {
            if (tcp_in_quickack_mode(tp)) {
                tcp_send_ack(sk);
            }  else {
                tcp_send_delayed_ack(sk);
            }
        }  else {
            __tcp_ack_snd_check(sk, 0);
        }
```

Since the data was already transferred, we delete the skb and call the data_ready callback to indicate that the socket is now ready for the next application read call.

```
no_ack:
        if (eaten)
            __kfree_skb(skb);
```

```
            else
                sk->data_ready(sk, 0);
            return 0;
        }
    }
```

The following code section is the slow path processing of received data segments. This is where we end up if this function was called internally from the kernel's "bottom half" or if prequeue processing couldn't proceed for some reason.

```
slow_path:
    if (len < (th->doff<<2) || tcp_checksum_complete_user(sk, skb))
        goto csum_error;
```

We do the PAWS check for out-of-order segments by checking for the timestamp TCP option.

```
    if (tcp_fast_parse_options(skb, th, tp) && tp->saw_tstamp &&
        tcp_paws_discard(tp, skb)) {
```

Even if PAWS checking indicates that we have received an out-of-order segment, we must still check for an incoming **RST**.

```
        if (!th->rst) {
            NET_INC_STATS_BH(PAWSEstabRejected);
            tcp_send_dupack(sk, skb);
            goto discard;
        }
    }
```

We resume standard slow path processing of data segments as specified by RFC 793. We must check the sequence number of all incoming packets.

```
    if (!tcp_sequence(tp, TCP_SKB_CB(skb)->seq, TCP_SKB_CB(skb)->end_seq)) {
```

If the incoming segment is not acceptable, send an acknowledgment.

```
        if (!th->rst)
            tcp_send_dupack(sk, skb);
        goto discard;
    }
```

If we receive an RST, we silently discard the segment.

```
    if(th->rst) {
        tcp_reset(sk);
        goto discard;
    }
    tcp_replace_ts_recent(tp, TCP_SKB_CB(skb)->seq);
    if (th->syn && !before(TCP_SKB_CB(skb)->seq, tp->rcv_nxt)) {
        TCP_INC_STATS_BH(TcpInErrs);
        NET_INC_STATS_BH(TCPAbortOnSyn);
        tcp_reset(sk);
        return 1;
```

```
    }
```

The fifth step for the **ESTABLISHED** state (page 72 in RFC 793) is to check the **ACK** field. The sixth step is to process the **URG** flag.

```
step5:
    if(th->ack)
        tcp_ack(sk, skb, FLAG_SLOWPATH);
    tcp_urg(sk, skb, th);
```

We are in the slow path where we haven't prequalified the segments, so we follow the steps in RFC 793. The seventh step on page 74 in the RFC is to process the segment data. Tcp_data_queue queues up data in the socket's normal receive queue. It puts segments that are out of order on the out_of_order_queue. For data in the socket's normal receive queue, processing is continued when the application program executes a read system call on the open socket, which will cause tcp_recvmsg to be called. The explanation of the socket-level receive is in Section 10.9.

```
    tcp_data_queue(sk, skb);
    tcp_data_snd_check(sk);
    tcp_ack_snd_check(sk);
    return 0;
csum_error:
    TCP_INC_STATS_BH(TcpInErrs);
discard:
    __kfree_skb(skb);
    return 0;
}
```

10.8 TCP TIME_WAIT State

Once one end of a connection receives an active close, it must stay in the **TIME_WAIT** state for two times the maximum segment lifetime. This section describes the processing of incoming packets when the TCP connection is in this state.

At this point, it is helpful to differentiate an active close from a passive close. An active close is caused by an explicit request by this end of the connection, but a passive close is done when a **FIN** is received from the peer. The **TIME_WAIT** state serves several purposes. One is to prevent old segments from a closed connection that are hanging around in the network from re-appearing in time to be confused with segments from a new connection. The other purpose is to hold on to the connection for a time that is longer than the maximum retransmission time, long enough to allow the peer to resend a last **ACK** (or **ACK** and data segment combination) when our last **ACK** was lost.

One of the problems of TCP/IP when implemented in large servers is that there could be substantial memory requirements for maintaining many sockets in the **TIME_WAIT** state. Although this is not a feature of primary importance to embedded systems designers, it illustrates how Linux TCP/IP is designed in part to meet the needs of TCP servers that maintain hundreds or thousands of open connections. The reader will remember that the primary vehicle for holding

TCP connections, receive buffers, and connection state is the sock structure. Each instance of a sock structure has memory requirements associated with it, including the sock structure itself and attached data buffers. In a very active server with many connections, it can get very expensive to hold many sockets active waiting for numerous connections to shut down. In the next section, we will see how Linux addresses this problem.

10.8.1 tcp_tw_bucket Structure

To reduce these requirements, the **TIME_WAIT** state processing does not use the sock structure. Instead, it uses a data structure that is smaller and has no attached receive buffers. This data structure is called the tcp_tw_bucket and it is defined in file *linux/include/net/tcp.h*. The tcp_tw_bucket shares the first 16 fields with the sock structure so it can use the same list maintenance pointers and functions.

```
struct tcp_tw_bucket {
```

The common part matches the sock structure.

```
    struct sock_common  __tw_common;
#define tw_family       __tw_common.skc_family
#define tw_state        __tw_common.skc_state
#define tw_reuse        __tw_common.skc_reuse
#define tw_bound_dev_if __tw_common.skc_bound_dev_if
#define tw_node         __tw_common.skc_node
#define tw_bind_node    __tw_common.skc_bind_node
#define tw_refcnt       __tw_common.skc_refcnt
```

Substate holds the states that are possible while processing a passive close, **FIN_WAIT_1**, **FIN_WAIT_2**, **CLOSING**, and **TIME_WAIT**.

```
    volatile unsigned char    tw_substate;
    unsigned char       tw_rcv_wscale;
```

The following fields are for socket de-multiplexing of incoming packets. These five fields are in the inet_opt structure.

```
    __u16               tw_sport;
    __u32               tw_daddr;
    __attribute__((aligned(TCP_ADDRCMP_ALIGN_BYTES)));
    __u32               tw_rcv_saddr;
    __u16               tw_dport;
    __u16               tw_num;
```

The fields from here to the end are unique to tcp_tw_bucket.

```
    int                 tw_hashent;
    int                 tw_timeout;
    __u32               tw_rcv_nxt;
__u32            tw_snd_nxt;
    __u32               tw_rcv_wnd;
    __u32               tw_ts_recent;
    long                tw_ts_recent_stamp;
```

```
    unsigned long       tw_ttd;
    struct tcp_bind_bucket    *tw_tb;
    struct hlist_node   *tw_death_node;
#if defined(CONFIG_IPV6) || defined(CONFIG_IPV6_MODULE)
    struct in6_addr     tw_v6_daddr;
    struct in6_addr     tw_v6_rcv_saddr;
    int                 tw_v6_ipv6only;
#endif
} ;
```

10.8.2 The tcp_timewait_state_process Function

Linux TCP supports fast time-wait recycling to prevent the number of connections in the **TIME_WAIT** state from using too many resources. Since the **TIME_WAIT** state can be maintained for several minutes, there is a possibility that the number of connections in the **TIME_WAIT** state can grow very large. Therefore, the time-wait buckets, tcp_tw_bucket structure, are maintained in slots accessed through a hash function. In a busy TCP server, the buckets are re-cycled depending on the system control value, tcp_tw_recycle, used.

The function tcp_timewait_state_process is in the file *linux/net/ipv4/tcp_minisocks.c*. It does most of the work of processing of incoming packets during **TIME_WAIT**. However, it also does the processing for the **FIN_WAIT_2** state, which is part of the active close but before the **TIME_WAIT** state in the TCP state machine. Tcp_timewait_state_process returns an enum, tcp_tw_status, defined in file *linux/include/net/tcp.h*. The values of the enum are shown in Table 10.2. If you examine the function closely, you may observe that it actually repeats most of the steps done by the function tcp_rcv_state_process, described earlier in this chapter, for most of the states **TIME_WAIT**, but in abbreviated form.

```
enum tcp_tw_status
tcp_timewait_state_process(struct tcp_tw_bucket *tw, struct sk_buff *skb,
                           struct tcphdr *th, unsigned len)
{
    struct tcp_opt tp;
    int paws_reject = 0;

    tp.saw_tstamp = 0;
```

Here we check to see if the incoming packet contained a timestamp. If it does, we do the PAWS check for an out-of-order segment.

```
    if (th->doff > (sizeof(struct tcphdr)>>2) && tw->ts_recent_stamp) {
        tcp_parse_options(skb, &tp, 0);
        if (tp.saw_tstamp) {
            tp.ts_recent = tw->tw_ts_recent;
            tp.ts_recent_stamp = tw->tw_ts_recent_stamp;
            paws_reject = tcp_paws_check(&tp, th->rst);
        }
    }
```

Similar to the tcp_rcv_state_process, we check to see if we are in the **FIN_WAIT2** state. If so, we must check for an incoming **FIN**, **ACK**, or an incoming out-of-order segment. If the earlier PAWS check found that the incoming segment is out of order, we must send an ACK.

```
    if (tw->substate == TCP_FIN_WAIT2) {
```

RFC 793 on page 69 states that an acknowledgment should be sent for an unacceptable segment if we are in **TIME_WAIT**, so we return TCP_TW_ACK to tell the caller to send an ACK

```
        if (paws_reject ||
             !tcp_in_window(TCP_SKB_CB(skb)->seq,
             TCP_SKB_CB(skb)->end_seq, tw->tw_rcv_nxt,
             tw->tw_rcv_nxt + tw->tw_rcv_wnd))
            return TCP_TW_ACK;
```

If the incoming segment contains an **RST**, we can finally kill off the connection.

```
        if (th->rst)
            goto kill;
```

If we receive a SYN but it is “old” or out of the window, we send a **RST**.

```
        if (th->syn && !before(TCP_SKB_CB(skb)->seq, tw->tw_rcv_nxt))
            goto kill_with_rst;
```

We check to see if the incoming segment has a duplicate **ACK**. If so, we will want to discard the segment.

```
        if (!after(TCP_SKB_CB(skb)->end_seq, tw->tw_rcv_nxt) ||
            TCP_SKB_CB(skb)->end_seq == TCP_SKB_CB(skb)->seq) {
            tcp_tw_put(tw);
            return TCP_TW_SUCCESS;
        }
```

If the arriving segment contains new data, we must send a reset to the peer to kill off the connection. We also stop the **TIME_WAIT** state timer.

```
        if (!th->fin || TCP_SKB_CB(skb)->end_seq != tw->rcv_nxt+1) {
kill_with_rst:
            tcp_tw_deschedule(tw);
            tcp_tw_put(tw);
            return TCP_TW_RST;
        }
```

At this point, we know the arriving segment is a **FIN** and we are still in the **FIN_WAIT_2** state, so we enter the actual **TIME_WAIT** state. We also process the received timestamp, saving the incoming timestamp value in ts_recent and marking when it was received.

```
        tw->tw_substate = TCP_TIME_WAIT;
        tw->tw_rcv_nxt = TCP_SKB_CB(skb)->end_seq;
        if (tp.saw_tstamp) {
            tw->ts_recent_stamp = xtime.tv_sec;
            tw->tw_ts_recent = tp.rcv_tsval;
        }
```

Tcp_tw_schedule is called to manage the timer, which determines the length of time the connection will remain in the **TIME_WAIT** state. RFC 1122 specifies that the timer should be 2MSL (two times the Maximum Segment Lifetime). If possible, Linux makes an attempt to reduce the time to an amount based on the RTO (Re-transmission Timeout). The comments contain a note apologizing for the IPv4-specific code, but it is OK because the IPv6 implementation, unlike IPv4, doesn't support fast time wait recycling. If we are part of an IPv6 connection, we pass a constant value of one minute to tcp_tw_schedule; otherwise, we pass in the RTO value, which could be less.

```
        if (tw->tw_family == AF_INET &&
            sysctl_tcp_tw_recycle && tw->tw_ts_recent_stamp &&
            tcp_v4_tw_remember_stamp(tw))
            tcp_tw_schedule(tw, tw->tw_timeout);
        else
            tcp_tw_schedule(tw, TCP_TIMEWAIT_LEN);
        return TCP_TW_ACK;
    }
```

Here we enter the "real" **TIME_WAIT** state. RFC 1122 states in section 4.2.2.13 that if we receive a **SYN** in a connection in the **TIME_WAIT** state, we may re-open the connection. However, we must assign the initial sequence number of the new connection to a value larger than the maximum sequence number used in the previous connection. We must also return to the **TIME_WAIT** state if the incoming SYN is a duplicate of an old one from the previous connection.

```
    if (!paws_reject &&
        (TCP_SKB_CB(skb)->seq == tw->tw_rcv_nxt &&
        (TCP_SKB_CB(skb)->seq == TCP_SKB_CB(skb)->end_seq || th->rst))) {
```

The value of zero in paws_reject indicates that the incoming segment is inside the window; therefore, it is either a **RST** or an **ACK** with no data.

```
        if (th->rst) {
```

It is possible that this incoming **RST** will result in **TIME_WAIT** Assassination (TWA). The system control, sysctl_tcp_rfc1337, can be set to prevent TWA. Linux TCP does not prevent TWA as its default behavior, so if the system control is not set, we kill off the connection [RFC1337].

```
            if (sysctl_tcp_rfc1337 == 0) {
kill:
                tcp_tw_deschedule(tw);
                tcp_tw_put(tw);
                return TCP_TW_SUCCESS;
             }
         }
```

The incoming segment must be a duplicate **ACK**, so we discard it. We also update the timer by calling tcp_tw_schedule.

```
        tcp_tw_schedule(tw, TCP_TIMEWAIT_LEN);
```

```
        if (tp.saw_tstamp) {
            tw->tw_ts_recent = tp.rcv_tsval;
            tw->tw_ts_recent_stamp = xtime.tv_sec;
        }
        tcp_tw_put(tw);
        return TCP_TW_SUCCESS;
    }
```

If we reached here, the PAWS test must have failed, so we must have either an out-of-window segment or a new **SYN**. All the out-of-window segments are acknowledged immediately. To accept a new **SYN**, it must not be an old duplicate. The check mandated by RFC 793 only works at slower network speeds, less than 40Mbit/second. Although, the PAWS checks is sufficient to ensure that we don't really need to check the sequence numbers, we do the mandated sequence number check.

```
    if (th->syn && !th->rst && !th->ack && !paws_reject &&
        (after(TCP_SKB_CB(skb)->seq, tw->tw_rcv_nxt) ||
        (tp.saw_tstamp && (s32)(tw->tw_ts_recent - tp.rcv_tsval) < 0))) {
        u32 isn = tw->tw_snd_nxt+65535+2;
        if (isn == 0)
            isn++;
        TCP_SKB_CB(skb)->when = isn;
        return TCP_TW_SYN;
    }
    if (paws_reject)
        NET_INC_STATS_BH(PAWSEstabRejected);
    if(!th->rst) {
```

We reset the **TIME_WAIT** state timer, but only if the incoming segment was an ACK or out of the window.

```
        if (paws_reject || th->ack)
            tcp_tw_schedule(tw, TCP_TIMEWAIT_LEN);
```

We tell the caller to acknowledge the bad segment.

```
        return TCP_TW_ACK;
    }
```

We must have received a **RST**, so we kill the connection.

```
    tcp_tw_put(tw);
    return TCP_TW_SUCCESS;
}
```

10.9 TCP Socket-Level Receive

When the user task is signaled that there is data waiting on an open socket, it calls one of the receive or read system calls on the open socket. These functions are translated at the socket layer into a call to tcp_recvmsg in file *linux/net/ipv4/tcp.c*. Tcp_recvmsg copies data from an open

socket into a user buffer. Comments in the code state that starting with Linux kernel version 2.3, the socket is locked.

```
int tcp_recvmsg(struct sock *sk, struct msghdr *msg,
                int len, int nonblock, int flags, int *addr_len)
{
    struct tcp_opt *tp = tcp_sk(sk);
    int copied = 0;
    u32 peek_seq;
    u32 *seq;
    unsigned long used;
    int err;
```

Target is set to the minimum number of bytes that this function should return.

```
    int target;
    long timeo;
    struct task_struct *user_recv = NULL;
    lock_sock(sk);
    TCP_CHECK_TIMER(sk);
    err = -ENOTCONN;
    if (sk->state == TCP_LISTEN)
        goto out;
    timeo = sock_rcvtimeo(sk, nonblock);
```

If the MSG_OOB flag is set, urgent data (incoming segments with the URG flag) is handled specially. At entry, seq is initialized to the next byte to be read because the copied_seq field of the tcp_opt structure contains the last byte that has been processed.

```
    if (flags & MSG_OOB)
        goto recv_urg;
    seq = &tp->copied_seq;
    if (flags & MSG_PEEK) {
        peek_seq = tp->copied_seq;
        seq = &peek_seq;
    }
```

The number of bytes to read, target, is set to the low-water mark for the socket, sk->rcvlowat or len, whichever is less. The MSG_WAITALL flag indicates whether this call will block for target number of bytes.

```
    target = sock_rcvlowat(sk, flags & MSG_WAITALL, len);
```

This do while loop is the main loop of tcp_recvmsg. In this loop, we will continue to copy bytes to the user until target bytes is reached or some other exception condition is detected while processing the incoming data segments.

```
    do {
        struct sk_buff * skb;
        u32 offset;
```

If after copying some data, we encounter urgent data while processing segments, we stop processing.

```
if (copied && tp->urg_data && tp->urg_seq == *seq)
    break;
```

Here we check to see if there is a signal pending on this socket to ensure the correct handling of the SIGURG signal. The comment in the code states: “FIXME: Need to check this doesn’t impact 1003.1g and move it down to the bottom of the loop.” Next, we check to see if the socket has timed out, in which case we return an error.

```
if (signal_pending(current)) {
    if (copied)
        break;
    copied = timeo ? sock_intr_errno(timeo) : -EAGAIN;
    break;
}
```

We get a pointer to the first buffer on the receive queue, and in the inner do while loop we walk through the receive queue until we find the first data segment. When we find the segment and know how many bytes to copy, we jump to found_ok_skb. Along the way, we calculate the number of bytes to copy (using the variable offset) from the skb.

```
skb = skb_peek(&sk->sk_receive_queue);
```

In this inner loop we keep examining packets until we find a valid data segment. Along the way, we check for **FIN** and **SYN**. If we see a **SYN**, we adjust the number of bytes to be copied by subtracting one from offset. A **FIN** drops us out of the loop.

```
do {
    if (!skb)
        break;
```

Now we check to see that the current byte to be processed is not before the first byte in the most recently received segment to see if somehow we got out of synchronization while processing the queue of packets. This is actually a redundant check because socket locking and multiple queues should prevent us from getting lost.

```
    if (before(*seq, TCP_SKB_CB(skb)->seq)) {
        printk(KERN_INFO "recvmsg bug: copied %X seq %X\ n",
               *seq, TCP_SKB_CB(skb)->seq);
        break;
    }
    offset = *seq - TCP_SKB_CB(skb)->seq;
    if (skb->h.th->syn)
        offset–;
    if (offset < skb->len)
        goto found_ok_skb;
    if (skb->h.th->fin)
        goto found_fin_ok;
    BUG_TRAP(flags&MSG_PEEK);
    skb = skb->next;
}  while (skb != (struct sk_buff *)&sk->sk_receive_queue);
```

If we get here, it means that we found nothing in the socket receive queue. If we have packets in the backlog queue, we try to process that, too.

```
        if (copied >= target && sk->backlog.tail == NULL)
            break;
```

Now we must make a few checks to see if we need to stop processing packets. We check for an error condition in the socket, if the socked is closed, or if we received a shutdown request from the peer (this would be a **RST** in a received packet, covered in Section 10.6).

```
        if (copied) {
            if (sk->sk_err ||
                sk->sk_state == TCP_CLOSE ||
                (sk->sk_shutdown & RCV_SHUTDOWN) ||
                !timeo ||
               (flags & MSG_PEEK))
                break;
        }  else {
            if (sock_flag(sk, SOCK_DONE))
                break;
            if (sk->err) {
                copied = sock_error(sk);
                break;
            }
            if (sk->shutdown & RCV_SHUTDOWN)
                break;
            if (sk->state == TCP_CLOSE) {
```

Done is set on a socket when a user closes, so normally it is nonzero when the connection state is **CLOSED**. If done is zero on a **CLOSED** socket, it means that an application program is trying to read from a socket that has never been connected, which, of course, is an error condition.

```
                if (!sock_flag(sk, SOCK_DONE)) {
                    copied = -ENOTCONN;
                    break;
                }
                break;
            }
            if (!timeo) {
                copied = -EAGAIN;
                break;
            }
        }
        cleanup_rbuf(sk, copied);
```

We are here because there are no more segments on the receive queue left to process. We will process any packets on the prequeue that have been pre-qualified for fast path processing. Previously, header prediction has indicated that there are likely to be data segments received when the connection state is **ESTABLISHED**. The prequeue packets are processed by the user task instead of in the context of the "bottom half." Ucopy.task is set to current, which forces the prequeue segments to be copied from the prequeue later by the user task, current.

```
        if (tp->ucopy.task == user_recv) {
```

Here we install a new reader task.

```
            if (!user_recv&& !(flags&(MSG_TRUNC|MSG_PEEK))) {
                user_recv = current;
                tp->ucopy.task = user_recv;
                tp->ucopy.iov = msg->msg_iov;
            }
            tp->ucopy.len = len;
            BUG_TRAP(tp->copied_seq == tp->rcv_nxt ||
                     (flags&(MSG_PEEK|MSG_TRUNC)));
```

If the prequeue is not empty, it must be processed before releasing the socket. If this is not done, the segment order will be damaged at the next iteration through this loop. The packet processing order on the receive side can be thought of as consisting of a series of four pseudo-queues, packets in flight, backlog, prequeue, and the normal receive queue. Each of these queues can be processed only if the packets ahead of it have already been processed. The receive queue is now empty, but the prequeue could have had segments added to it when the socket was released in the last iteration through this loop. The prequeue is processed at the do_prequeue label.

```
            if (skb_queue_len(&tp->ucopy.prequeue))
                goto do_prequeue;
        }
        if (copied >= target) {
```

Now, the backlog is processed to see if it is possible to do any direct copying of packets on that queue. At this point, we have processed the prequeue. Release_sock walks through all the packets on the backlog queue, sk->backlog, before waking up any tasks waiting on the socket.

```
            release_sock(sk);
            lock_sock(sk);
        }  else {
```

If there is no more data to copy and absolutely nothing more to do, we sit and wait for more data. The function tcp_data_wait puts the socket and (therefore the calling task) in the wait state, TASK_INTERRUPTIBLE. Daniel Bovet and Marco Cesati have an excellent discussion of Linux process scheduling policy in Chapter 11 [BOVET02].

```
            timeo = tcp_data_wait(sk, timeo);
        }
        if (user_recv) {
            int chunk;
```

We account for any data directly copied from the backlog queue in the previous step. We also return the scheduler to its normal state.

```
            if ((chunk = len - tp->ucopy.len) != 0) {
                NET_ADD_STATS_USER (TCPDirectCopyFromBacklog, chunk);
                len -= chunk;
                copied += chunk;
            }
            if (tp->rcv_nxt == tp->copied_seq &&
                skb_queue_len(&tp->ucopy.prequeue)) {
```

```
do_prequeue:
```

This is where we jumped to process any packets on the prequeue. The function tcp_prequeue_process does the work and is covered in Section 10.5.2. After calling, we adjust chunk to account for any data copied from the prequeue.

```
                tcp_prequeue_process(sk);
                if ((chunk = len - tp->ucopy.len) != 0) {
                    NET_ADD_STATS_USER (TCPDirectCopyFromPrequeue, chunk);
                    len -= chunk;
                    copied += chunk;
                }
            }
        }
        if ((flags & MSG_PEEK) && peek_seq != tp->copied_seq) {
            if (net_ratelimit())
                printk(KERN_DEBUG
                       "TCP(%s:%d): Application bug, race in MSG_PEEK.\ n",
                       current->comm, current->pid);
                peek_seq = tp->copied_seq;
        }
        continue;
```

This is where we jumped from the previous inner loop when we found a data segment on the receive queue. We figure out how much data we have to copy from the len field of skb and offset calculated earlier.

```
    found_ok_skb:
        used = skb->len - offset;
        if (len < used)
            used = len;
```

We must check for urgent data. Unless the socket option, SO_OOBINLINE, was set (indicated by the urginline field of the sock structure), we skip the urgent data because it was processed separately.

```
        if (tp->urg_data) {
            u32 urg_offset = tp->urg_seq - *seq;
            if (urg_offset < used) {
                if (!urg_offset) {
                    if (!sock_flag(sk, SOCK_URGINLINE)) {
                        ++*seq;
                        offset++;
                        used–;
                        if (!used)
                            goto skip_copy;
                    }
                }  else
                    used = urg_offset;
            }
        }
```

This is where we copy the data to user space. If we get an error while copying, we return an EFAULT.

```
        if (!(flags&MSG_TRUNC)) {
            err = skb_copy_datagram_iovec(skb, offset, msg->msg_iov, used);
            if (err) {
```

This is an exception condition.

```
                if (!copied)
                    copied = -EFAULT;
                break;
            }
        }
        *seq += used;
        copied += used;
        len -= used;
    skip_copy:
        if (tp->urg_data && after(tp->copied_seq,tp->urg_seq)) {
            tp->urg_data = 0;
```

Now that we are done processing urgent data, we turn on the fast path (set TCP header prediction). Fast path processing was turned off if input processing encountered a segment with urgent data (**URG** flag on) and a valid urgent pointer field in the TCP header.

```
            tcp_fast_path_check(sk, tp);
        }
        if (used + offset < skb->len)
            continue;
        if (skb->h.th->fin)
            goto found_fin_ok;
        if (!(flags & MSG_PEEK))
            tcp_eat_skb(sk, skb);
        continue;
```

This is where we jumped if we found a packet containing a **FIN** in the receive queue. RFC 793 says that we must count the **FIN** as one byte in the sequence and TCP window calculation.

```
    found_fin_ok:
        ++*seq;
        if (!(flags & MSG_PEEK))
            tcp_eat_skb(sk, skb);
        break;
    }  while (len > 0);
```

We are at the end of the outer while loop, processing socket buffers until we have copied the amount of data, len, requested by the caller in the application program. If we dumped out of the loop, leaving any data on the prequeue, it must be processed now before getting out.

```
    if (user_recv) {
        if (skb_queue_len(&tp->ucopy.prequeue)) {
            int chunk;
            tp->ucopy.len = copied > 0 ? len : 0;
            tcp_prequeue_process(sk);
            if (copied > 0 && (chunk = len - tp->ucopy.len) != 0) {
                NET_ADD_STATS_USER (TCPDirectCopyFromPrequeue, chunk);
                len -= chunk;
```

```
                copied += chunk;
            }
        }
        tp->ucopy.task = NULL;
        tp->ucopy.len = 0;
    }
```

Cleanup_rbuf cleans up the TCP receive buffer. It will send an **ACK** if necessary.

```
    cleanup_rbuf(sk, copied);
    TCP_CHECK_TIMER(sk);
    release_sock(sk);
    return copied;
```

We are done, so release the socket and get out.

```
out:
    TCP_CHECK_TIMER(sk);
    release_sock(sk);
    return err;
```

We jumped here if we encountered urgent data while processing segments. Tcp_recv_urg copies the urgent data to the user.

```
recv_urg:
    err = tcp_recv_urg(sk, timeo, msg, len, flags, addr_len);
    goto out;
}
```

10.9.1 Receiving Urgent Data

Urgent data is data received in a segment that has the **URG** TCP flag and a valid urgent pointer. Urgent data, also known as *Out-of-Band* (OoB) data, is handled separately from the normal data in the data segments. Theoretically, it is handled as a higher priority and gets passed up to the socket as OoB data. The function tcp_recv_urg in file *linux/net/ipv4/tcp.c* is called from tcp_recvmsg when a segment containing urgent data is encountered while processing the stream of data segments.

```
static int tcp_recv_urg(struct sock * sk, long timeo,
                        struct msghdr *msg, int len, int flags,
                        int *addr_len)
{
    struct tcp_opt *tp = tcp_sk(sk);
```

The SOCK_URGINLINE flag in the sock structure is set from the SO_OOBINLINE socket option, which states that urgent data should be handled as if it were ordinary segment data. If this socket option was set, it is an error because we are in the function that is supposed to process the urgent data specially.

```
    if (sock_flag(sk, SOCK_URGINLINE)|| !tp->urg_data ||
        tp->urg_data == TCP_URG_READ)
        return -EINVAL;
```

```
    if (sk->state==TCP_CLOSE && !sock_flag(sk, SOCK_DONE))
        return -ENOTCONN;
```

Now that we have survived the initial steps, we copy the urgent data into the user's buffer.

```
    if (tp->urg_data & TCP_URG_VALID) {
        int err = 0;
        char c = tp->urg_data;
        if (!(flags & MSG_PEEK))
            tp->urg_data = TCP_URG_READ;
```

Setting the MSG_OOB flag tells the application that urgent data has been received. Memcpy_toiovec does the actual copying.

```
        msg->msg_flags|=MSG_OOB;

        if(len>0) {
            if (!(flags & MSG_TRUNC))
                err = memcpy_toiovec(msg->msg_iov, &c, 1);
            len = 1;
        }  else
            msg->msg_flags|=MSG_TRUNC;
        return err ? -EFAULT : len;
    }
    if (sk->state == TCP_CLOSE || (sk->sk_shutdown & RCV_SHUTDOWN))
        return 0;
```

We should not block in this call, regardless of the blocking state of the socket. All implementations as well as BSD have the same behavior.

```
    return -EAGAIN;
}
```

10.10 Summary

In this chapter, we discussed the receive side of TCP and UDP. This chapter is a companion to Chapter 8, in which we discussed the sending side. We covered the receive-side packet handling in UDP and the socket hash de-multiplexing. We also covered processing of incoming broadcast and multicast packets. For TCP, we discussed input state handling. In addition, we look at how received packets are processed while in the **ESTABLISHED** state, and the **TIME_WAIT** state.

Chapter 11: Internet Protocol Version 6 (IPv6)

In this chapter, we examine IPv6. We introduce some basics about the protocol and how it enhances IPv4. In this chapter, as in the rest of the book, we concentrate on the internals of TCP/IP. We discuss how the IPv6 protocol works in the Linux kernel networking environment. For historical reasons, some of the Linux networking facilities are IPv4 specific. This chapter covers the areas where IPv6 has its own separate facilities, even though they are similar to facilities in IPv4. This is because that, as of the writing of this book, the two protocol stacks, IPv4 and IPv6, were not entirely integrated. A few features of IPv6 are still implemented separately from those used with IPv4. For example, the IPv6 Forwarding Information Base (FIB) is unique to IPv6. This is because the IPv4 FIB has network address fields that don't support the longer address type for IPv6.

There is active ongoing development of IPv6, particularly by the USAGI project [USAGI03]. The author has made an attempt to find the most recent stable release of IPv6 for Linux 2.6. See Chapter 1, "Introduction," and Section 1.1 for specific information about the source code revision used in this book. In addition, the companion CD-ROM contains a copy of all the sources referenced in this book, including the IPv6 source discussed in this chapter.

11.1 Introduction

The Internet protocol is 24 years old at the time of this writing. It was originally specified in 1980 [RFC 760]. In the early days of IP, it was never envisioned how popular the Internet would become. The most fundamental limitation associated with IPv4 is its limited address size. IPv4 addresses are 32-bits long, limiting the total number of addresses. Back in the early 1990s, when there began to be dramatic increases in the use of the Internet, it was becoming clear that the address space was too restricted. There were simply not enough addresses available because of the 4-byte address in IPv4. In the early 1990s, people started to think about ways to expand the address space. Most organizations on the Internet were assigned network IDs from the class C address space, and this only made three bytes available. An innovation called Classless Inter-domain Routing (CIDR) was specified in 1993 [RFC 1519] that freed up more network IDs by removing restrictions of the class-based addressing scheme. IPv4 addressing is covered in more detail in Chapter 3, "TCP/IP in Embedded Systems." In the mid 1990s, IPv6 was specified. [RFC 1883] IPv6 increases the address length to 128 bits, which should be enough for practically every person, place, or thing to have a unique address. As of 2003, the world is still largely running IPv4, but IPv6 is starting to enjoy wider use. Linux IPv6 is becoming the focus of attention for those working on IPv6 protocols and utilities.

11.2 Facilities in IPv6

There is much more to IPv6 than a longer address. In this section, we outline some of these significant capabilities. Later as we discuss the IPv6 sources, we will show where in the source code these facilities are implemented.

Besides the 128-bit address, IPv6 includes the ability to autoconfigure addresses. When using IPv4, we have to statically assign an address to a machine before putting it on the network.

Another alternative often used with IPv4 is the Dynamic Host Configuration Protocol (DHCP), which can assign an address when the computer comes online. However, IPv6 supports a mechanism called *stateless auto-configuration* where a node coming online can be identified directly by its link-local address. (As we will see in Section 11.3, the link-local address is based on the machine's unique MAC or Ethernet address.) This means that any device can be plugged in to a local IPv6 net without having to use DHCP or other means of preconfiguring an address. This concept allows the Ethernet or other link layer address to be built into IPv6 128-bit addresses.

IPv6 has a method where hosts and routers can automatically find each other. This mechanism replaces the ARP protocol in IPv4. We know that with IPv6, devices can autoconfigure themselves by self-assigning unique IPv6 addresses (called *link-local addresses)* based on their MAC addresses. Instead of ARP, IPv6 provides a method where the routers can find each of the hosts automatically, machines can automatically determine their default router, and hosts can find each other. This mechanism is called *Neighbor Discovery*, (ND). The ND mechanism consists of four message types: router advertisement, router solicitation, neighbor advertisement, and neighbor solicitation [RFC 2461]. In Linux, ND is based on the neighbor cache facility which was discussed in detail in Chapter 6.

Routing based on Quality-of-Service (QoS) routing is done in IPv4 by using traffic class-based routing. With this method, the routing entries in the Routing Policy Database (RPDB) include the packet's IP header ToS field. When a packet arrives at the queuing layer, forwarding decisions are based on the QoS as defined by the route. We covered this process in Chapter 9, "The Network Layer, IP," in detail. In contrast, with IPv6, QoS is built into the protocol specification. The IP header has a Flow Label field intended to be used by routers to make "hard" routing decisions based on desired QoS and Linux includes support for IPv6 flows.

IP mobility is part of IPv6. A node can introduce itself into a new network based on a permanently assigned IPv6 address. It also supports jumbograms, which are very large packets. IPv6 has a much different header format than IPv4 does. In addition, IPv6 has security built in.

11.3 IPv6 Addressing

IPv6 specifies several basic types of addresses: *unicast*, *anycast*, and *multicast*. With IPv6, addresses are applied to network interfaces, not nodes as is the case with IPv4. An IPv6 unicast address defines a single interface. A packet sent to a unicast address will be delivered to the particular network interface which corresponds to the address. In contrast, an anycast address defines a set of network interfaces where generally each network interface belongs to a different machine. A packet sent to an anycast address will be delivered to one of the interfaces and generally, the closest one as determined by the address scope. Multicast addresses also define a set of nodes. They are similar in concept to IPv4 multicast addresses. A packet sent to a multicast address will be delivered to each interface belonging to the set also known as a group. IPv6 has no broadcast address type. As we shall see, the broadcast address type of IPv4 is superceded by the IPv6 all nodes link-local and node-local scope multicast address.

IPv4 addresses are usually expressed in dotted decimal notation. In contrast, IPv6 addresses are expressed in a string of eight hexadecimal values separated by the colon character [RFC 2373].

```
x:x:x:x:x:x:x:x
```

Each x represents a 16-bit value. Often, IPv6 addresses will consists of multiple segments of zeros. To make it easier to express, sequential groups of zeros can be represented with a double colon,::. Only one double colon can appear in an address. Leading zeros on a 16-bit group can be dropped, but trailing zeros cannot be dropped. An example of an address representing a unicast address could be as follows:

```
1080:0:0:0:8:800:200C:417A
```

This same address could be written as shown here, where the :: replaces the 0:0:0. This is called the *compressed form*.

```
1080::8:800:200C:417A
```

This address is the unspecified address.

```
0:0:0:0:0:0:0:0
```

It could also be written as:

```
::
```

For another example, we can look at this multicast address.

```
FF01:0:0:0:0:0:0:101
```

The same address can be written like this.

```
FF01::101
```

There is also a type of notation where IPv4 address can be shown in a mixed notation. Each X represents a 16-bit hexadecimal portion, and each D is an 8-bit decimal number [RFC 3493].

```
X:X:X:X:X:X:D:D:D:D
```

For example, an IPv4 mapped IPv6 address is as follows.

```
0:0:0:0:0:FFFF:10.1.5.10
```

In compressed notation, it would be written like this.

```
::FFFF:10.1.5.10
```

IPv4 has a way of representing the network portion and the host portion of an address. The network portion can be shown as a netmask or is often shown in CIDR notation. IPv6 has a similar concept. For example, the following address shows a unicast address with its 56-bit netmask.

```
11AB::DE40:FEDC:BA98:7654:3210/56
```

The network number for this address is the following.

```
11AB:0:0:DE00:0:0:0:0
```

It can be shown a different way.

```
11AB::DE00:0:0:0:0
```

The host portion is like this.

```
40:FEDC:BA98:7654:3210
```

IPv6 also defines a loopback address. All but the trailing bit are zeros.

```
::1
```

It defines an address called the *unspecified address*, which is all zeros.

```
::
```

IPv4 would use the first three leading bits of the address to define the class of the address. Refer to Chapter 3 for more on IPv4 addressing. IPv6 also uses a variable group of leading bits to define the type of address. This leading group is called the *Format Prefix* (FP). Table 11.1 shows the address space allocation according to the format prefixes.

Table 11.1: IPv6 Address Type Allocation

Format Prefix in Binary	**Fraction of Address Space**	**Allocation**
0000 0000	1/256	Reserved. The unspecified address (all zeros) and the embedded IPv4 addresses are assigned out of this space.
0000 0001	1/256	Unassigned
0000 001	1/128	Reserved for NSAP allocation
0000	1/128	Reserved for IPX allocation
0000 011	1/128	Unassigned
0000 1	1/32	Unassigned
0001	1/16	Unassigned
001	1/8	Aggregatable global unicast Addresses
010	1/8	Unassigned
011	1/8	Unassigned
100	1/8	Unassigned
101	1/8	Unassigned

Table 11.1: IPv6 Address Type Allocation

Format Prefix in Binary	Fraction of Address Space	Allocation
110	1/8	Unassigned
1110	1/16	Unassigned
1111 0	1/32	Unassigned
1111 10	1/64	Unassigned
1111 110	1/128	Unassigned
1111 1110 0	1/512	Unassigned
1111 1110 10	1/1024	Link-local unicast addresses
1111 1110 11	1/1024	Site-local unicast addresses
1111 1111	1/256	Multicast addresses

IPv4 uses the ARP protocol to map IP address to link layer MAC addresses. In contrast, IPv6 does not use ARP; instead, it uses a concept called *link-local addresses*. The link-local addresses incorporate the 6-byte Ethernet address directly into the IPv6 addresses. The link-local addresses are prefaced by the bits FE8. The address space allocation for link-local addresses is shown in Table 11.1. For example, the 802.11 wireless adapter on this laptop has the following Ethernet address.

```
00-02-2D-84-D1-A1
```

The link-local IPv6 address for the adapter would be as follows.

```
FE80::2:2D84:D1A1
```

By definition, IPv6 addresses are assigned to the network interface, not to the host. In addition, an interface might have multiple addresses of any type. However, at least one link-local address must be assigned to each interface.

There are a few other specific multicast addresses used with Neighbor Discovery and other protocols. The first is the solicited-node multicast address [RFC 1883]. This address is formed by taking the lower 24 bits of either an unicast or anycast address and applying the following 96-bit prefix, FF02:0:0:0:0:1. This yields a range of multicast addresses between FF02::1:0:0 and FF02::1:FFFF:FFFF. The next is the all-routers multicast address, which is the link-local scoped address for reaching all nodes, FF02::1. Another is the all-nodes multicast address to reach all link-local scoped nodes, FF02::1.

Linux provides a union data structure to represent the IPv6 address type, defined in *file* *in6.h*.

```
struct in6_addr
{
    union
    {
        __u8            u6_addr8[16];
```

```
        __u16           u6_addr16[8];
        __u32           u6_addr32[4];
    }  in6_u;
#define s6_addr         in6_u.u6_addr8
#define s6_addr16       in6_u.u6_addr16
#define s6_addr32       in6_u.u6_addr32
} ;
```

This structure allows the address to be specified as either a sequence of 32-bit words, 16-bit segments, or 8-bit bytes. The following two definitions, also in *in6.h*, define types for the unspecified address format (wildcard address) and the loopback address, both of which are frequently referenced in the IPv6 sources.

```
#define IN6ADDR_ANY_INIT {  {  {  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 }  }  }
#define IN6ADDR_LOOPBACK_INIT {  {  {  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1 }  }  }
```

There is also new socket address structure IPv6 addresses, shown in Chapter 5, "Linux Sockets." The mapped IPv4 form of address can be specified through an IPv6 socket.

11.4 IPv6 Packet Format

The IPv6 packet is simplified. Many of the fields that were required in the IPv4 header are optional in the IPv6 header. These fields are now in optional headers called *extension headers*. There is a next header field in the main IP header and in all the extension headers. The next header field contains a specific value indicating the type of the next extension header if it exists. If there is no next extension header, the next header field contains the type for the first upper layer header, or as shown in Table 11.2, the specific value NEXTHDR_NONE if there are no more extension headers. Figure 11.1 shows the IPv6 main header. All the headers are in the same packet as the main header. However, since the extension headers are optional, the overhead is reduced because headers that are not necessary can be left out of the packet.

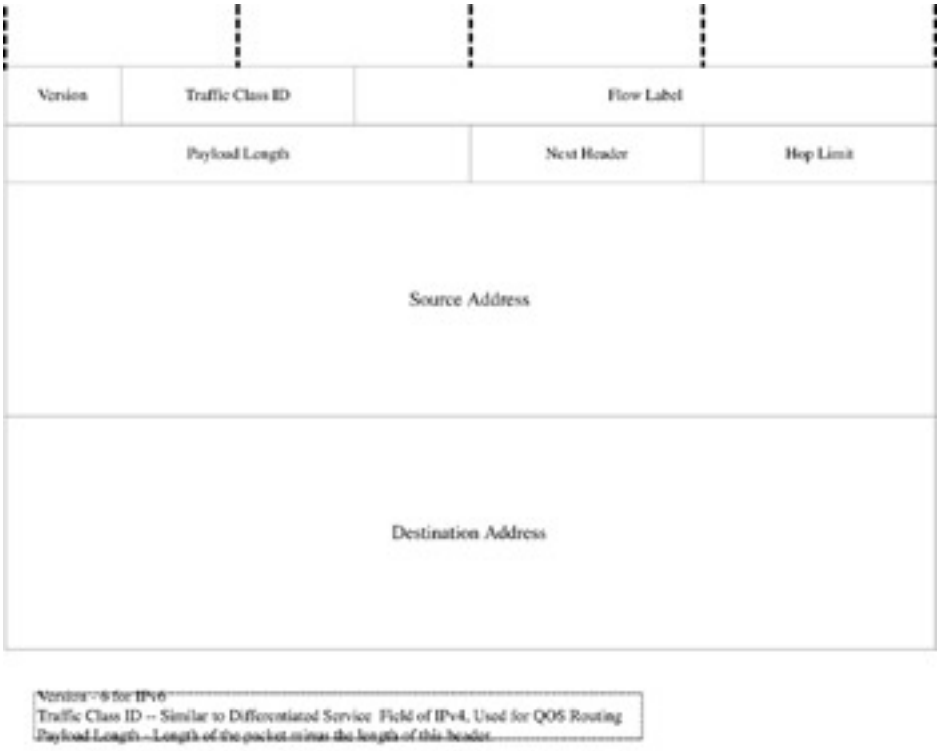
Version
Traffic Class ID
Flow Label
Payload Length
Next Header
Hop Limit
Source Address
Destination Address
Version – 6 for IPv6
Traffic Class ID – Similar to Differentiated Service Field of IPv4, Used for QOS Routing
Payload Length – Length of the packet minus the length of this header

Figure 11.1: IPv6 header.

Table 11.2: Next Header Field Values

Name	Value	Header Description
NEXTHDR_HOP	0	The hop-by-hop option header.
NEXTHDR_TCP	6	The next header is a TCP header.
NEXTHDR_UDP	17	The next header is a UDP header.
NEXTHDR_IPV6	41	The next header is for IPv6 in IPv6 tunnel.
NEXTHDR_ROUTING	43	The next header is the routing header.
NEXTHDR_FRAGMENT	44	Header for fragmentation or reassembly.
NEXTHDR_ESP	50	Header for Encapsulating Security Payload (ESP).
NEXTHDR_AUTH	51	This is for the authentication header.
NEXTHDR_ICMP	58	IPv6 ICMP.
NEXTHDR_NONE	59	This value means that there is no next header.
NEXTHDR_DEST	60	This is the destination options header.
NEXTHDR_MAX	255	This is the maximum value for the next-header field.

The extensions headers that follow the IPv6 header have a recommended order, shown here.

- Hop-by-hop options header
- Destination options header

- Routing header
- Fragment header
- Upper layer header

Table 11.2 shows the values for the next header field in the IPv6 header. These values are defined in file *ipv6.h*.

The format for the fragmentation header is shown in Figure 11.2. Its structure, frag_hdr, is defined in file *ipv6.h*.

Figure 11.2: The fragmentation header.

11.5 IPv6 Implementation in Linux

The IPv6 protocol is implemented in the kernel as a module. IPv6 uses many similar mechanisms to IPv4. The protocol connects with the socket layer in the same way as IPv4 does. In addition, it contains a protocol switch table for transport layer de-multiplexing. Therefore, it uses the same method as IPv4 for dispatching incoming packets to UDP, TCP, and other protocols.

To support IP packet de-multiplexing, each of the next header field values has IP protocol numbers. These numbers are defined in file *linux/include/linux/in6.h*.

```
#define IPPROTO_HOPOPTS      0     /* IPv6 hop-by-hop options       */
#define IPPROTO_ROUTING     43     /* IPv6 routing header           */
#define IPPROTO_FRAGMENT    44     /* IPv6 fragmentation header     */
#define IPPROTO_ICMPV6      58     /* ICMPv6                        */
#define IPPROTO_NONE        59     /* IPv6 no next header           */
#define IPPROTO_DSTOPTS     60     /* IPv6 destination options      */
```

Linux IPv6 has a structure called the inet6_dev structure. This structure is similar to the inet_dev structure for IPv4 discussed in Chapter 6. The inet6_dev structure, defined in file *linux/include/net/if_inet6.h*, contains the nondevice-specific information for a particular network interface. This is where the interface address list and the multicast address list is maintained for the network interface device.

```
struct inet6_dev
{
```

Most of the fields are similar to the inet_dev structure for IPv4. Dev points back to the network interface device. Addr_list is the list of addresses for this interface, and mc_list is the list of multicast addresses that this interface has joined.

```
    struct net_device   *dev;
    struct inet6_ifaddr *addr_list;
    struct ifmcaddr6    *mc_list;
```

```
    struct ifmcaddr6    *mc_tomb;
    rwlock_t            mc_lock;
    unsigned long       mc_v1_seen;
    unsigned long       mc_maxdelay;
    unsigned char       mc_qrv;
    unsigned char       mc_gq_running;
    unsigned char       mc_ifc_count;
```

The following field is the general query timer, and the next field is the interface change timer.

```
    struct timer_list   mc_gq_timer;
    struct timer_list   mc_ifc_timer;
    struct ifacaddr6    *ac_list;
    rwlock_t            lock;
    atomic_t            refcnt;
    __u32               if_flags;
    int                 dead;

#ifdef CONFIG_IPV6_PRIVACY
    u8                  rndid[8];
    u8                  entropy[8];
    struct timer_list   regen_timer;
    struct inet6_ifaddr *tempaddr_list;
    __u8                work_eui64[8];
    __u8                work_digest[16];
#endif

    struct neigh_parms  *nd_parms;
    struct inet6_dev    *next;
    struct ipv6_devconf cnf;
    struct ipv6_devstat stats;
} ;
```

11.5.1 IPv6 Initialization

IPv6 is initialized in the file *linux/net/ipv6/af_ine6t.c*. The functions in this file provide most of the glue for the protocols in the AF_INET6 address family, including socket registration. The main initialization function is inet6_init.

```
static int __init inet6_init(void);
```

Inet6_init is very similar to the function, inet_init used for IPv4 initial ization. In this function, we essentially follow the same steps as inet_init. First, we create three slab caches for the three socket types, TCP6, UDP6, and RAW. We register the RAW socket type first by calling inet6_register_protosw. This is different from inet_init, which registers all three socket types at one time. Next, we do the basic socket registration of the AF_INET6 family calling sock_register.

```
(void) sock_register(&inet6_family_ops);
```

Now, applications can create RAW sockets, but hopefully, no one will try to create a UDP or TCP socket before we complete the registration process. Now, we initialize the IPv6 MIBs and register ourselves with the sysctl facility if it is configured in this kernel. We initialize the ICMP

protocol by calling icmpv6_init, the neighbor discovery protocol by calling ndisc_init, and the IGMP protocol by calling igmp6_init.

Next, we set up our entries in the /proc file system. IPv6 uses the standard /proc facility like many other facilities and protocols in the Linux kernel. We register with the filesystem in the same way that IPv4 and other protocols do. The following directories in /proc are unique for IPv6.

```
if_inet6
raw6
anycast6
ip6_flowlabel
igmp6
mcfilter6
snmp6
rt6_stats
sockstat6
```

Now that /proc is set up, we can initialize some of the functionality unique to IPv6. At the end, the inet6_init performs the following steps. The next function is called to initialize the IPv6 notifier chains. The IPv6 notifier chain is built on the generic notifier facility discussed in Chapter 4, "Linux Networking Interfaces and Device Drivers."

```
ipv6_netdev_notif_init();
```

We call this function to register the packet handler function and the link layer protocol type so we can receive IPv6 packets.

```
ipv6_packet_init();
```

We initialize IPv6 routing facility. This is discussed in Section 11.5.

```
ip6_route_init();
```

Ip6_flowlabel_init initializes flow labels, which is the method in IPv6 of routing packets based on QoS. Next, address autoconfiguration is initialized by addrconf_init. The next function initializes the Simple Internet Transition (SIT), which is the IPv6 over IPv4 tunnel device.

```
ip6_flowlabel_init();
addrconf_init();
sit_init();
```

Next, by calling the following four functions, we initialize the handlers for the IPv6 extension headers. Earlier, we discussed how IPv6 consists of a basic header and a number of extension headers. Each extension header type has a registered protocol handler just like UDP and TCP.

```
ipv6_rthdr_init();
ipv6_frag_init();
ipv6_nodata_init();
ipv6_destopt_init();
```

Finally, we initialize the two transport protocols, UDP and TCP. Each of these initialization functions registers a protocol handler just like IPv4 UDP and TCP.

```
udpv6_init();
tcpv6_init();
```

As discussed earlier in this book, transport layer de-multiplexing of incoming packets is done by calling a protocol handler function through the destination cache entry. The handler functions are placed in a linked list through a registration process. In IPv6, we do the protocol registration in a similar fashion to IPv4. However, IPv6 adds a flags field to its version of the registration structure, inet6_protocol, defined in the file *linux/include/net/policy.h.*

```
struct inet6_protocol
{
```

As is the case with the inet_protocol structure used with IPv4, two handler functions are defined: one for incoming error packets and one for incoming normal packets.

```
    int   (*handler)(struct sk_buff **skb, unsigned int *nhoffp);

    void  (*err_handler)(struct sk_buff *skb,
           struct inet6_skb_parm *opt,
           int type, int code, int offset,
           __u32 info);
```

The flags field is unique to IPv6. In recent versions of IPv6, the value in flags determines if a security check is made on the incoming packet. This is a way to see if packets should be rejected because of a failed security association.

```
    unsigned int  flags;
} ;
```

The values in the flags field of the inet6_protocol structure are defined as follows. The FINAL value says that there is no XFRM policy and therefore there is no need to do a security check of this particular input packet. XFRM is for IP encapsulated packets such as IPSec.

```
#define INET6_PROTO_FINAL     0x2
```

However, the value NOPOLICY says that a check should be made in the SADB to see if there is a transformation policy associated with this incoming packet.

```
#define INET6_PROTO_NOPOLICY  0x1
```

One of the steps in the initialization of the IPv6 protocol is to register the packet handler with the link layer IPV6 protocol type, ETH_P_IPV6, which is defined as 0x86DD. The function ipv6_packet_init in file *linux/net/ipv6/ipv6_sockglue.c* does this.

```
void __init ipv6_packet_init(void)
{
    dev_add_pack(&ipv6_packet_type);
}
```

The structure ipv6_packet_type is initialized as the following also in the same file.

```
static struct packet_type ipv6_packet_type = {
    .type = __constant_htons(ETH_P_IPV6),
    .func = ipv6_rcv,
} ;
```

This is so the function ipv6_rcv is registered as the low-level packet handler for IPv6 packets when they are received at the queuing layer.

11.6 IPv6 Socket Implementation

IPv6 sockets are accessed by specifying the IPv6 address family, AF_INET6. There are some socket API changes for IPv6 sockets, and these are shown in Chapter 5. It is important to point out that IPv6 semantics require that the socket programming API be protocol independent. To support this, Linux sockets are implemented in such a way that the definitions of socket addresses are entirely protocol independent. The fact that IPv4 or IPv6 protocols are used should be entirely transparent to the application programmer. Other than a few updates for version 2.6, there are no implementation changes in the generic socket implementation itself because, as shown in Chapter 5, it is already entirely protocol independent. The socket structure is defined to work with both address families simultaneously.

As we recall from earlier chapters, the proto_ops structure maps the transport layer internal functions to the socket function calls. As we see from the following definitions of the function mappings, there is not much change from IPv4. The proto_ops structure for IPv6 (AF_INET6 address family) SOCK_DGRAM type sockets are shown next. This structure is initialized in file *linux/net/ipv6/af_inet6.c*.

```
struct proto_ops inet6_dgram_ops = {
        .family =       PF_INET6,
        .owner =        THIS_MODULE,
```

The release and bind functions are unique to IPv6.

```
        .release =      inet6_release,
        .bind =         inet6_bind,
        .connect =      inet_dgram_connect,
        .socketpair =   sock_no_socketpair,
        .accept =       sock_no_accept,
```

The getname function is new to support the new address mapping capabilities.

```
        .getname =      inet6_getname,
        .poll =         datagram_poll,
```

The ioctl function must be different for IPv6.

```
        .ioctl =        inet6_ioctl,
        .listen =       sock_no_listen,
        .shutdown =     inet_shutdown,
```

```
        .setsockopt =   inet_setsockopt,
        .getsockopt =   inet_getsockopt,
        .sendmsg =      inet_sendmsg,
        .recvmsg =      inet_recvmsg,
        .mmap =         sock_no_mmap,
        .sendpage =     sock_no_sendpage,
} ;
```

The SOCK_STREAM socket mapping in proto_ops is not much different from IPv4either. The functions that are not unique to the individual member protocols in AF_INET6 address family are defined in file *linux/net/ipv6/af_inet.c*.

```
struct proto_ops inet6_stream_ops = {
        .family =       PF_INET6,
        .owner =        THIS_MODULE,
        .release =      inet6_release,
        .bind =         inet6_bind,
        .connect =      inet_stream_connect,
        .socketpair =   sock_no_socketpair,
        .accept =       inet_accept,
        .getname =      inet6_getname,
        .poll =         tcp_poll,
        .ioctl =        inet6_ioctl,
        .listen =       inet_listen,
        .shutdown =     inet_shutdown,
        .setsockopt =   inet_setsockopt,
        .getsockopt =   inet_getsockopt,
        .sendmsg =      inet_sendmsg,
        .recvmsg =      inet_recvmsg,
        .mmap =         sock_no_mmap,
        .sendpage =     tcp_sendpage
} ;
```

Even though the preceding mappings look like the ones in IPv4, when the user opens an AF_INET6 socket, SOCK_STREAM packets are processed by the IPv6 version of TCP, and SOCK_DGRAM packets are processed by the IPv6 version of UDP. This is because IPv6 registered its own unique versions of the transport protocols with the AF_INET6 layer. Transmitted packets are sent through either IPv4 or IPv6 by the transport layer send message function.

As we recall from Chapter 5, the sock structure contained a union with protocol-specific areas for each of the protocols using sockets. For IPv6, this area is defined by the ipv6_pinfo structure.

```
struct ipv6_pinfo {
    struct in6_addr     saddr;
    struct in6_addr     rcv_saddr;
    struct in6_addr     daddr;
    struct in6_addr     *daddr_cache;

    __u32               flow_label;
    __u32               frag_size;
    int                 hop_limit;
    int                 mcast_hops;
    int                 mcast_oif;
```

The following packed structure is for the packet option flags.

```
    union {
        struct {
            __u8  srcrt:2,
                  rxinfo:1,
                  rxhlim:1,
                  hopopts:1,
                  dstopts:1,
                  authhdr:1,
                  rxflow:1;
        }  bits;
        __u8      all;
    }  rxopt;
```

The following field is for the socket option flags.

```
    __u8                mc_loop:1,
                        recverr:1,
                        sndflow:1,
                        pmtudisc:2,
                        ipv6only:1;

    struct ipv6_mc_socklist   *ipv6_mc_list;
    struct ipv6_ac_socklist   *ipv6_ac_list;
    struct ipv6_fl_socklist   *ipv6_fl_list;
    __u32                     dst_cookie;

    struct ipv6_txoptions     *opt;
    struct sk_buff            *pktoptions;
    struct {
        struct ipv6_txoptions *opt;
        struct rt6_info       *rt;
        int                   hop_limit;
    }  cork;
} ;
```

11.7 IPv6 Fragmentation and De-Fragmentation Implementation

IPv6 fragmentation is implemented in the file *linux/net/ipv6/ip6_output.c*. Typically, we know that we have to fragment output packets when the next-hop MTU is smaller than the fragment size. This was previously determined by ip6_output, as we'll show in Section 11.8, when it called the function ip6_fragment.

```
static int ip6_fragment(struct sk_buff *skb, int (*output)(struct sk_buff*))
...
```

In ip6_fragment, we have a slow path and a fast path. The fast path is used if the sk_buff for the output packet already points to a list of prepared fragments as determined by the macro skb_shinfo(skb)->frag_list.

First we look at the fast path. The fast path would be likely to be used if the MTU for the route was already known at the time the socket was set up for a particular TCP connection. Now, we walk through the list of fragments. Each fragment must be of sufficient size and not shared. If all is OK, we prepare the fragment header in the following code snippet.

```
. . .
        tmp_hdr = kmalloc(hlen, GFP_ATOMIC);
        if (!tmp_hdr) {
            IP6_INC_STATS(Ip6FragFails);
            return -ENOMEM;
        }
        *prevhdr = NEXTHDR_FRAGMENT;
        memcpy(tmp_hdr, skb->nh.raw, hlen);
        __skb_pull(skb, hlen);
        fh = (struct frag_hdr*)__skb_push(skb, sizeof(struct frag_hdr));
        skb->nh.raw = __skb_push(skb, hlen);
        memcpy(skb->nh.raw, tmp_hdr, hlen);
        ipv6_select_ident(skb, fh);
        fh->nexthdr = nexthdr;
        fh->reserved = 0;
        fh->frag_off = htons(IP6_MF);
        frag_id = fh->identification;

        first_len = skb_pagelen(skb);
        skb->data_len = first_len - skb_headlen(skb);
        skb->len = first_len;
        skb->nh.ipv6h->payload_len =
               htons(first_len - sizeof(struct ipv6hdr));

        for (;;) {
```

Here we prepare the header of the next frame before the previous one is transmitted.

```
            if (frag) {
                frag->h.raw = frag->data;
                fh = (struct frag_hdr*)__skb_push(frag,
                          sizeof(struct frag_hdr));
                frag->nh.raw = __skb_push(frag, hlen);
                memcpy(frag->nh.raw, tmp_hdr, hlen);
                offset += skb->len - hlen - sizeof(struct frag_hdr);
                fh->nexthdr = nexthdr;
                fh->reserved = 0;
                fh->frag_off = htons(offset);
                if (frag->next != NULL)
                    fh->frag_off |= htons(IP6_MF);
                fh->identification = frag_id;
                frag->nh.ipv6h->payload_len =
                   htons(frag->len - sizeof(struct ipv6hdr));
                ip6_copy_metadata(frag, skb);
            }
```

This is where we put the skb on the sending queue.

```
            err = output(skb);
            if (err || !frag)
```

```
                break;
            skb = frag;
            frag = skb->next;
            skb->next = NULL;
        }

        if (tmp_hdr)
            kfree(tmp_hdr);

        if (err == 0) {
            IP6_INC_STATS(Ip6FragOKs);
            return 0;
        }

        while (frag) {
            skb = frag->next;
            kfree_skb(frag);
            frag = skb;
        }

        IP6_INC_STATS(Ip6FragFails);
        return err;
    }
. . .
```

Now we look at the slow path. The slow path involves splitting the skb into fragments before placing each fragment on the output queue.

IPv6 de-fragmentation is implemented in the file *linux/net/ipv6/reassembly.c*. To do de-fragmentation, a queue of input fragments is maintained. In addition, there is a timer associated with each entry on the queue. Orphaned fragments are aged out and discarded if they remain unclaimed on the queue beyond the reasonable lifetime of a packet [RFC 1883].

Since IPv6 fragments are in optional headers that follow the main IPv4 header, we actually register a separate protocol within the AF_INET6 address family to handle the fragment header type. This protocol is like a transport protocol in the sense that its handler is dispatched when the protocol number, IPPROTO_FRAGMENT, is encountered in the next header field of the IP header. Here we show the structure for registering the fragment protocol's handler function with the AF_INET6 family.

```
static struct inet6_protocol frag_protocol =
{
```

IPv6_frag_rcv is the handler function. Flags is set to indicate that there is no security check required for this packet type.

```
    .handler     =      ipv6_frag_rcv,
    .flags       =      INET6_PROTO_NOPOLICY,
} ;
```

The fragment protocol is initialized by ipv6_frag_init.

```
void __init ipv6_frag_init(void)
```

```
{
    if (inet6_add_protocol(&frag_protocol, IPPROTO_FRAGMENT) < 0)
        printk(KERN_ERR "ipv6_frag_init: Could not register protocol\ n");

. . .
}
```

The fragment receive handler, ipv6_frag_rcv, also in the same file, is executed when the IPPROTO_FRAGMENT is encountered in the next header field of the IPv6 packet.

```
static int ipv6_frag_rcv(struct sk_buff **skbp, unsigned int *nhoffp)
...
```

After checking the fragment header for validity, we attempt to find a fragment in the existing fragment queue with the same identification value.

```
    if ((fq = fq_find(fhdr->identification, &hdr->saddr, &hdr->daddr))
!= NULL) {
        int ret = -1;

        spin_lock(&fq->lock);
```

If we can't find a matching fragment, we add the new one to the queue.

```
        ip6_frag_queue(fq, skb, fhdr, *nhoffp);
```

Then we call ip6_frag_reasm to try to put the fragments together.

```
        if (fq->last_in == (FIRST_IN|LAST_IN) &&
            fq->meat == fq->len)
            ret = ip6_frag_reasm(fq, skbp, nhoffp, dev);

        spin_unlock(&fq->lock);
        fq_put(fq);
        return ret;
    }
. . .
}
```

11.8 IPv6 Output

IP output processing in IPv6 is very similar to IPv4. Once a packet is ready to be transmitted, the destination cache entry's output field will point to the function ip6_output, which is in the file *linux/net/ipv6/ip6_output.c*.

```
int ip6_output(struct sk_buff *skb)
{
```

As we can see, this function does little more than decide whether to fragment the packet. This determination is done by checking the route MTU value in the destination cache entry, dst.

```
    if ((skb->len > dst_pmtu(skb->dst) || skb_shinfo(skb)->frag_list))
        return ip6_fragment(skb, ip6_output2);
    else
        return ip6_output2(skb);
}
```

The function ip6_output2 in the same file continues the processing of output packets.

```
int ip6_output2(struct sk_buff *skb);
```

It doesn't do much more than check if the destination is a multicast address. If it is, the skb is cloned and a copy is sent to the loopback address. As of the time of this writing, actual multicast routing was not supported in Linux IPv6. Before this function exits, it calls ip6_output_finish in the same file to finish the processing of output packets.

The function ip6_output_finish completes the processing of the output packets.

```
static inline int ip6_output_finish(struct sk_buff *skb)
{

    struct dst_entry *dst = skb->dst;
    struct hh_cache *hh = dst->hh;
```

At this point, what we do is very much like IPv4 output processing. We can see now how the neighbor system in Linux TCP/IP was designed with IPv6 in mind. If the hardware header cache is defined, we call the hh_output function, which will queue up the packet to the network interface device. If not, we call the output function defined for the neighbor in the destination cache, dst.

```
    if (hh) {
        int hh_alen;

        read_lock_bh(&hh->hh_lock);
        hh_alen = HH_DATA_ALIGN(hh->hh_len);
        memcpy(skb->data - hh_alen, hh->hh_data, hh_alen);
        read_unlock_bh(&hh->hh_lock);
                        skb_push(skb, hh->hh_len);
        return hh->hh_output(skb);
    }  else if (dst->neighbour)
        return dst->neighbour->output(skb);
    kfree_skb(skb);
    return -EINVAL;

}
```

11.9 IPv6 Input

As shown earlier, we register a specific function as a packet handler for IPv6 packets and the handler gets called when a network interface device receives a packet of type ETH_P_IPV6. This function in many ways is simpler than its equivalent for IPv4 because of the simpler IPv6 header format. The IPv6 header is simpler because, unlike IPv4, all the IPv6 options are in

separate headers. The packet handling function, ipv6_rcv, is defined in file *linux/net/ipv6/ip6_input.c* and is called from the softirq context, or "bottom half" in the packet queuing layer.

```
int ipv6_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type
*pt)
{
    struct ipv6hdr *hdr;
    u32            pkt_len;
    if (skb->pkt_type == PACKET_OTHERHOST)
        goto drop;
    IP6_INC_STATS_BH(Ip6InReceives);
    if ((skb = skb_share_check(skb, GFP_ATOMIC)) == NULL) {
        IP6_INC_STATS_BH(Ip6InDiscards);
        goto out;
    }
```

Here we store the index of the incoming network interface device. We don't refer to the actual network interface device, dev, once the packet is queued.

```
    ((struct inet6_skb_parm *)skb->cb)->iif = dev->ifindex;
```

We must check to make sure that the incoming packet is at least long enough to hold the IPv6 header.

```
    if (skb->len < sizeof(struct ipv6hdr))
        goto err;
    if (!pskb_may_pull(skb, sizeof(struct ipv6hdr)))
        goto drop;
    hdr = skb->nh.ipv6h;
```

We check to make sure that the version field is six for IPv6.

```
    if (hdr->version != 6)
        goto err;
```

We get the payload length directly from the packet header. The payload length includes the length of the optional headers if there are any.

```
    pkt_len = ntohs(hdr->payload_len);
```

It is possible that the payload length is zero if our input packet is a jumbogram.

```
    if (pkt_len || hdr->nexthdr != NEXTHDR_HOP) {
        if (pkt_len + sizeof(struct ipv6hdr) > skb->len)
            goto truncated;
        if (pkt_len + sizeof(struct ipv6hdr) < skb->len) {
            if (__pskb_trim(skb, pkt_len + sizeof(struct ipv6hdr)))
                goto drop;
            hdr = skb->nh.ipv6h;
            if (skb->ip_summed == CHECKSUM_HW)
                skb->ip_summed = CHECKSUM_NONE;
        }
```

```
    }
```

If the next header is a hop-by-hop header, then we call ipv6_parse_hopopts to extract the hop options from the incoming packet. The hop options are stored back in the socket buffer, skb.

```
    if (hdr->nexthdr == NEXTHDR_HOP) {
        unsigned int nhoff = offsetof(struct ipv6hdr, nexthdr);
    skb->h.raw = (u8*)(hdr+1);
        if (ipv6_parse_hopopts(&skb, &nhoff) < 0) {
            IP6_INC_STATS_BH(Ip6InHdrErrors);
            return 0;
        }
    }
```

Now we continue processing by calling ip6_rcv_finish, which calls ip_route_input to set up the destination cache for this incoming packet by setting the dst field in the skb.

```
    return NF_HOOK(PF_INET6,NF_IP6_PRE_ROUTING, skb, dev,
                   NULL, ip6_rcv_finish);
truncated:
    IP6_INC_STATS_BH(Ip6InTruncatedPkts);
err:
    IP6_INC_STATS_BH(Ip6InHdrErrors);
drop:
    kfree_skb(skb);
out:
    return 0;
}
```

At this point, the processing is very similar to IPv4 input packet processing. Input packets may have a local destination in this machine or they may need to be forwarded. The packet destination was calculated when the ip_route_input function was called during the first stage of input packet processing. For example, if the destination is an internal destination on a machine, the input field in the dst_entry will point to the function ip6_input, which continues the processing of the input packets. Next, ip6_input calls ip_input_finish to deliver the packet to the upper layer protocols. If the destination is a locally reachable host, output will point to the function dev_queue_xmit, which queues the output packet to the network interface drivers. This process is the same as IPv4 and is covered in earlier chapters.

```
static inline int ip6_input_finish(struct sk_buff *skb)
. . .
```

Next, the rest of the processing is done by the input functions for each of the optional headers (other than the hop-by-hop header, which was processed already). The nexthdr field may contain the protocol number for an optional header or it may be UDP, TCP, IGMP, ICMP, or perhaps some other protocol.

```
    hash = nexthdr & (MAX_INET_PROTOS - 1);
    if ((ipprot = inet6_protos[hash]) != NULL) {
        int ret;

        smp_read_barrier_depends();
```

```
        if (ipprot->flags & INET6_PROTO_FINAL) {
            if (!cksum_sub && skb->ip_summed == CHECKSUM_HW) {
                skb->csum = csum_sub(skb->csum,
                                     csum_partial(skb->nh.raw,
                                                  skb->h.raw-skb->nh.raw,
                                                  0));
                cksum_sub++;
            }
        }
        if (!(ipprot->flags & INET6_PROTO_NOPOLICY) &&
              !xfrm6_policy_check(NULL, XFRM_POLICY_IN, skb))
            goto discard;
```

Here is where we call the handler function for the protocol defined in the nexthdr field of the IP header.

```
        ret = ipprot->handler(&skb, &nhoff);
        if (ret > 0)
            goto resubmit;
        else if (ret == 0)
            IP6_INC_STATS_BH(Ip6InDelivers);
    }  else {
        if (!raw_sk) {
            if (xfrm6_policy_check(NULL, XFRM_POLICY_IN, skb)) {
                IP6_INC_STATS_BH(Ip6InUnknownProtos);
                icmpv6_param_prob(skb, ICMPV6_UNK_NEXTHDR, nhoff);
            }
        }  else {
            IP6_INC_STATS_BH(Ip6InDelivers);
            kfree_skb(skb);
        }
    }
. . .
```

11.10 IPv6 UDP

IPv6 has its own implementation of the UDP protocol, in the file *linux/net/ipv6/udp.c*. Although the source code for IPv6 is mostly separate from IPv4, all the socket structures are shared, including the dynamic state information. There are no duplicate sockets or data structures outside of the protocol families. As with IPv4, UDP has an initialization function that is called from the IP initialization function as shown earlier. The initialization function for UDP is udpv6_init.

```
void __init udpv6_init(void)
{
```

As is the case with the IPv4 function, inet_add_protocol registers the packet handlers.

```
    if (inet6_add_protocol(&udpv6_protocol, IPPROTO_UDP) < 0)
        printk(KERN_ERR "udpv6_init: Could not register protocol\ n");
    inet6_register_protosw(&udpv6_protosw);
}
```

The udpv6_protocol structure is shown here. The handler for incoming UDP packets is udpv6_rcv.

```
static struct inet6_protocol udpv6_protocol = {
    .handler      =     udpv6_rcv,
    .err_handler  =     udpv6_err,
```

The flags field is initialized to allow XFRM transformations to be used with incoming UDP packets. This will allow packet security checks with the SDB.

```
    .flags        =     INET6_PROTO_NOPOLICY|INET6_PROTO_FINAL,
} ;
```

The inet_protosw structure for registration is also defined in file *ipv6/udp.c*.

```
static struct inet_protosw udpv6_protosw = {
    .type        =      SOCK_DGRAM,
    .protocol    =      IPPROTO_UDP,
```

The prot and the ops structures for IPv6 UDP are shown here.

```
    .prot        =      &udpv6_prot,
    .ops         =      &inet6_dgram_ops,
```

Capability has to do with permissions. As is the case with IPv4 UDP, anybody has permission to send UDP datagrams.

```
    .capability   =    -1,
    .no_check     =    UDP_CSUM_DEFAULT,
    .flags        =    INET_PROTOSW_PERMANENT,
} ;
```

The UDP proto functions for IPv6 are defined as follows. This structure maps the socket calls to the protocol-specific functions.

```
struct proto udpv6_prot = {
    .name =       "UDP",
    .close =      udpv6_close,
    .connect =    udpv6_connect,
```

The following two functions are the only ones shared with IPv4. It is interesting to note that although the socket level ioctl is different for SOCK_DGRAM type sockets, the protocol level ioctl is the same as UDP for IPv4.

```
    .disconnect = udp_disconnect,
    .ioctl =      udp_ioctl,
    .destroy =    udpv6_destroy_sock,
    .setsockopt = udpv6_setsockopt,
    .getsockopt = udpv6_getsockopt,
    .sendmsg =    udpv6_sendmsg,
    .recvmsg =    udpv6_recvmsg,
```

```
    .backlog_rcv =udpv6_queue_rcv_skb,
    .hash =       udp_v6_hash,
    .unhash =     udp_v6_unhash,
    .get_port =   udp_v6_get_port,
} ;
```

We noted earlier how IPv6 has its own implementation of UDP. The separate sources are required for a couple of reasons. One is that the different address format requires different address comparison logic. Address comparisons are required when doing a lookup to find the destination socket for an incoming packet. A second reason is we need the ability to send and receive messages with AF_INET type destination addresses through IPv6 sockets, and this requires some special handling.

The function, udp6_sendmsg is called when the application wants to transmit a message of any type from a SOCK_DGRAM type socket of the AF_INET6 protocol family. Another reason why IPv6 requires a separate set of functions for UDP is to support flow labeling, which is a fundamental feature of IPv6 for supporting QoS. Although some data structures for IPv4 such as flowi are updated for version 2.6, the standard Linux framework does not support flow labeling. We should look at a small section of code in udp6_sendmsg to see how it decides to send packets via the IPv4 protocol.

```
static int udpv6_sendmsg(struct kiocb *iocb, struct sock *sk, struct
msghdr *msg, int len)
. . .
```

The variable sin6 points to the "name" part of the msghdr structure, which is in the form of a sockaddr_in6 structure.

```
    if (sin6) {
        if (addr_len < offsetof(struct sockaddr, sa_data))
            return -EINVAL;
        switch (sin6->sin6_family) {
        case AF_INET6:
            if (addr_len < SIN6_LEN_RFC2133)
                return -EINVAL;
            daddr = &sin6->sin6_addr;
            break;
```

Here we check for the address families showing that it is legal to receive a message sent to an AF_INET type address family destination through an AF_INET6 socket.

```
        case AF_INET:
            goto do_udp_sendmsg;
        case AF_UNSPEC:
            msg->msg_name = sin6 = NULL;
            msg->msg_namelen = addr_len = 0;
            daddr = NULL;
            break;
        default:
            return -EINVAL;
        }
    }  else if (!up->pending) {
        if (sk->sk_state != TCP_ESTABLISHED)
```

```
            return -EDESTADDRREQ;
        daddr = &np->daddr;
    }  else
        daddr = NULL;
. . .
do_udp_sendmsg:
            if (__ipv6_only_sock(sk))
                return -ENETUNREACH;
```

We call udp_sendmsg, which is the ordinary IPv4 UDP send message function when we are sending datagrams to AF_INET type destinations.

```
            return udp_sendmsg(iocb, sk, msg, len);
```

11.11 IPv6 TCP

IPv6 also has its own version of the TCP protocol, which can be found in file *linux/net/ipv6/tcp_ipv6.c*. The initialization procedure for TCP is the same as UDP. Its handler functions are defined as follows.

```
static struct inet6_protocol tcpv6_protocol = {
```

Tcp_v6_rcv is the handler function for incoming TCP segments.

```
    .handler      =     tcp_v6_rcv,
    .err_handler  =     tcp_v6_err,
```

As with UDP, we allow security policy checks of incoming packets using the XFRM mechanism.

```
    .flags        =     INET6_PROTO_NOPOLICY|INET6_PROTO_FINAL,
} ;
```

The proto structure for TCP is defined below. Most of the fields are mapped to the very same functions used with IPv4, but we have noted a few exceptions here.

```
struct proto tcpv6_prot = {
    .name         =     "TCPv6",
    .close        =     tcp_close,
```

The connect function is different for IPv6. This is to handle mapped IPv4 addresses.

```
    .connect      =     tcp_v6_connect,
    .disconnect   =     tcp_disconnect,
    .accept       =     tcp_accept,
    .ioctl        =     tcp_ioctl,
```

Socket initialization and destruction functions also are unique for IPv6. This is because there are a few fields in the sock structure that are initialized differently from IPv4.

```
    .init         =     tcp_v6_init_sock,
```

```
    .destroy      =     tcp_v6_destroy_sock,
    .shutdown     =     tcp_shutdown,
    .setsockopt   =     tcp_setsockopt,
    .getsockopt   =     tcp_getsockopt,
    .sendmsg      =     tcp_sendmsg,
    .recvmsg      =     tcp_recvmsg,
```

Finally, the backlog receive function is different. All the socket lookup and socket hash functions are different, too. Both the address and port are used to look up a socket, so all these functions must be implemented specially for the longer address format in IPv6.

```
    .backlog_rcv  =     tcp_v6_do_rcv,
    .hash         =     tcp_v6_hash,
    .unhash       =     tcp_unhash,
    .get_port     =     tcp_v6_get_port,
} ;
```

In general, TCP is implemented separately to support IPv6 addressing and the ability to specify IPv4 destinations over IPv6 sockets. We will look at a few code snippets from some of the TCP functions to see how IPv6 handles things differently. For example, let's look at the TCP connect function for IPv6, tcp_v6_connect.

```
static int tcp_v6_connect(struct sock *sk, struct sockaddr *uaddr, int
addr_len)
. . .
```

Here we check to see if the application has requested any specific flow labeling for this socket. We recall from the earlier discussion that flow labeling is the IPv6 method of QoS. The variable np points to the IPv6 protocol-specific info in the sock structure.

```
    if (np->sndflow) {
        fl.fl6_flowlabel = usin->sin6_flowinfo&IPV6_FLOWINFO_MASK;
        IP6_ECN_flow_init(fl.fl6_flowlabel);
        if (fl.fl6_flowlabel&IPV6_FLOWLABEL_MASK) {
            struct ip6_flowlabel *flowlabel;
            flowlabel = fl6_sock_lookup(sk, fl.fl6_flowlabel);
            if (flowlabel == NULL)
                return -EINVAL;
            ipv6_addr_copy(&usin->sin6_addr, &flowlabel->dst);
            fl6_sock_release(flowlabel);
        }
    }
```

If the application wants to connect to INADDR_ANY, that means that we really want to connect to the loopback addresses. Therefore, we explicitly set the bit to define the loopback address type.

```
    if(ipv6_addr_any(&usin->sin6_addr))
        usin->sin6_addr.s6_addr[15] = 0x1;

    addr_type = ipv6_addr_type(&usin->sin6_addr);
```

We don't support multicast addresses for TCP.

```
    if(addr_type & IPV6_ADDR_MULTICAST)
        return -ENETUNREACH;
```

Here we check to see if the address type is link-local.

```
    if (addr_type&IPV6_ADDR_LINKLOCAL) {
        if (addr_len >= sizeof(struct sockaddr_in6) &&
            usin->sin6_scope_id) {
```

The field sin6_scope_id field is the scope of the link-local address. It is used to specify the network interface device index. If the socket is bound to an interface, we check to see if the socket is bound to the same interface through which the application is requesting the connection. If not, the request is invalid.

```
            if (sk->sk_bound_dev_if &&
                sk->sk_bound_dev_if != usin->sin6_scope_id)
                return -EINVAL;

            sk->sk_bound_dev_if = usin->sin6_scope_id;
        }
```

Here we check to see if our socket is bound to an interface. This is because, by definition, a link-local destination requires that an interface be known.

```
        if (!sk->sk_bound_dev_if)
            return -EINVAL;
    }

    if (tp->ts_recent_stamp &&
        ipv6_addr_cmp(&np->daddr, &usin->sin6_addr)) {
        tp->ts_recent = 0;
        tp->ts_recent_stamp = 0;
        tp->write_seq = 0;
    }

    ipv6_addr_copy(&np->daddr, &usin->sin6_addr);
    np->flow_label = fl.fl6_flowlabel;
```

Now, we handle TCP over IPv4. If the address to which we are trying to connect is an IPv4 mapped address, we want the TCP from IPv4 to handle packets for this socket.

```
    if (addr_type == IPV6_ADDR_MAPPED) {
        u32 exthdrlen = tp->ext_header_len;
        struct sockaddr_in sin;
        SOCK_DEBUG(sk, "connect: ipv4 mapped\ n");
        if (__ipv6_only_sock(sk))
            return -ENETUNREACH;
        sin.sin_family = AF_INET;
        sin.sin_port = usin->sin6_port;
        sin.sin_addr.s_addr = usin->sin6_addr.s6_addr32[3];
```

We set the backlog receive function to the correct one for IPv4 TCP and then call the IPv4 TCP connect function with the IPv4 portion of the destination address.

```
        tp->af_specific = &ipv6_mapped;
        sk->sk_backlog_rcv = tcp_v4_do_rcv;
        err = tcp_v4_connect(sk, (struct sockaddr *)&sin, sizeof(sin));
        if (err) {
            tp->ext_header_len = exthdrlen;
            tp->af_specific = &ipv6_specific;
            sk->sk_backlog_rcv = tcp_v6_do_rcv;
            goto failure;
        }  else {
            ipv6_addr_set(&np->saddr, 0, 0, htonl(0x0000FFFF),
                          inet->saddr);
            ipv6_addr_set(&np->rcv_saddr, 0, 0, htonl(0x0000FFFF),
                          inet->rcv_saddr);
        }

        return err;
    }
. . .
```

11.12 ICMPV6

The ICMP protocol for IPv6 is similar to ICMP for IPv4, but it has a few significant changes. For example, the destination unreachable, echo request, and echo reply messages are similar [RFC 2463]. In addition, the ICMPv6 implementation is similar to ICMPv4 discussed in Chapter 9. The function icmpv6_rcv is the handler for ICMPv6 packets, IPPROTO_ICMPV6. It is implemented in file *linux/net/ipv6/icmp.c*.

```
static int icmpv6_rcv(struct sk_buff **pskb, unsigned int *nhoffp)
. . .
```

We will skip the preliminaries and show how each type of ICMPv6 packet is processed.

```
    switch (type) {
    case ICMPV6_ECHO_REQUEST:
```

This is no different from IPv4; we simply send an echo reply.

```
        icmpv6_echo_reply(skb);
        break;

    case ICMPV6_ECHO_REPLY:
```

We don’t need to do anything special for an incoming echo reply.

```
        break;
    case ICMPV6_PKT_TOOBIG:
```

There is a note in the comments that suggests that we should update the destination cache if the packet contained a router header. However, the current version of the destination cache won’t support this.

```
        if (!pskb_may_pull(skb, sizeof(struct ipv6hdr)))
```

```
            goto discard_it;
        hdr = (struct icmp6hdr *) skb->h.raw;
        orig_hdr = (struct ipv6hdr *) (hdr + 1);
```

If the packet was 2 bit, we try to do a path MTU discovery, which hopefully will reduce the size of the MTU for this route.

```
        rt6_pmtu_discovery(&orig_hdr->daddr, &orig_hdr->saddr, dev,
                           ntohl(hdr->icmp6_mtu));
```

Here we drop through to do the notification. By notifying, we will inform the application of the problem via the open socket.

```
    case ICMPV6_DEST_UNREACH:
    case ICMPV6_TIME_EXCEED:
    case ICMPV6_PARAMPROB:
        icmpv6_notify(skb, type, hdr->icmp6_code, hdr->icmp6_mtu);
        break;
```

The next four packet types are neighbor discover packets. These are covered in Section 11.13.

```
    case NDISC_ROUTER_SOLICITATION:
    case NDISC_ROUTER_ADVERTISEMENT:
    case NDISC_NEIGHBOUR_SOLICITATION:
    case NDISC_NEIGHBOUR_ADVERTISEMENT:
    case NDISC_REDIRECT:
        ndisc_rcv(skb);
        break;
```

In IPv6, the multicast group management reports and queries are defined as ICMP messages [RFC 2710]. When we receive these messages we forward to the “IGMP” protocol for IPv6. These message types are discussed in Section 11.14.

```
    case ICMPV6_MGM_QUERY:
        igmp6_event_query(skb);
        break;

    case ICMPV6_MGM_REPORT:
        igmp6_event_report(skb);
        break;

    case ICMPV6_MGM_REDUCTION:
        break;

    case ICMPV6_NI_QUERY:
    case ICMPV6_NI_REPLY:
        break;

    default:
        if (net_ratelimit())
            printk(KERN_DEBUG "icmpv6: msg of unknown type\ n");
```

This type is informational.

```
        if (type & ICMPV6_INFOMSG_MASK)
            break;
```

We have an unknown error. We will notify the upper layers.

```
        icmpv6_notify(skb, type, hdr->icmp6_code, hdr->icmp6_mtu);
. . .
```

11.13 Neighbor Discovery

The Neighbor Discovery (ND) protocol is used by hosts and routers for mutual discovery on locally connected nets [RFC 2461]. This protocol replaces two protocols in IPv4. One of these protocols, no longer needed in IPv6, is ARP. In IPv4, ARP was used to map an IP address to a link layer address. As we saw in Section 11.3, this is no longer necessary because IPv6 addresses include a link-local address type in which the link-layer addresses are built in to the IPv6 address itself. Another facility that sometimes used IPv4 was router discovery [RFC 1256]. Router discovery was a protocol in which routers would send out periodic advertisements with advertised addresses and preferences. Hosts could choose a default gateway from among the advertisements.

We covered the generic neighbor system in Chapter 6. In that chapter, we discussed how the neighbor system was designed for the ND protocol, even though it is used for an ARP cache in IPv4. We create a local instance of the generic neighbor table. In the *ndisc.c*, the neighbor table nd_tbl is initialized.

```
struct neigh_table nd_tbl = {
. . .
```

The constructor and proxy constructor member functions are set as follows.

```
    .constructor =      ndisc_constructor,
    .pconstructor =     pndisc_constructor,
    .pdestructor =      pndisc_destructor,
    .proxy_redo =       pndisc_redo,
. . .
}
```

We also initialize a neighbor operations structure for the generic operations, the hardware header operations, and the direct operations. In each case, most of the member functions use the generic functions defined for the neighbor system, but the family field is initialized to AF_INET6. For example, this is how the ndisc_generic_ops structure is initialized for the IPv6 ND protocol.

```
static struct neigh_ops ndisc_generic_ops = {
    .family =           AF_INET6,
    .solicit =          ndisc_solicit,
    .error_report =     ndisc_error_report,
```

The following four functions are the same ones that are used for the IPv4 neighbor cache.

```
    .output =           neigh_resolve_output,
    .connected_output = neigh_connected_output,
    .hh_output =        dev_queue_xmit,
    .queue_xmit =       dev_queue_xmit,
} ;
```

In the hardware header operations, the only local function is the solicit member function.

```
    .solicit =          ndisc_solicit,
```

ND messages are defined as part of the ICMPv6 protocol. Therefore, there is no packet handler registered specifically for ND. The protocol includes four different message types, defined in file *linux/include/net/ndisc.h*. These values are shown in Table 11.3.

Table 11.3: Neighbor Discovery Messages

Message Type	**Value**
NDISC_ROUTER_SOLICITATION	133
NDISC_ROUTER_ADVERTISEMENT	134
NDISC_NEIGHBOUR_SOLICITATION	135
NDISC_NEIGHBOUR_ADVERTISEMENT	136
NDISC_REDIRECT	137

When an ND message is received by ICMP, it calls the function ndisc_rcv implemented in file *linux/net/ipv6/ndisc.c*.

```
int ndisc_rcv(struct sk_buff *skb)
{
```

Ndisc_rcv extracts the message from the packet and decodes it.

```
. . .

    switch (msg->icmph.icmp6_type) {
```

As of this writing, there isn't support for the router solicitation message NDISC_ROUTER_SOLICITATION.

```
    case NDISC_NEIGHBOUR_SOLICITATION:
        ndisc_recv_ns(skb);
        break;
    case NDISC_NEIGHBOUR_ADVERTISEMENT:
        ndisc_recv_na(skb);
        break;
    case NDISC_ROUTER_ADVERTISEMENT:
        ndisc_router_discovery(skb);
        break;
    case NDISC_REDIRECT:
        ndisc_redirect_rcv(skb);
        break;
```

```
    } ;
. . .
}
```

The neighbor solicitation message is sent for two reasons. The first is to discover the link-layer address for a connected neighbor. The other is to determine the reachability status of a neighbor. The function ndisc_recv_ns in file *ndisc.c* processes the neighbor solicitation messages.

```
static void ndisc_recv_ns(struct sk_buff *skb);
```

The first thing we do is some validity checks. If the source address is the unspecified address, IPV6_ADDR_ANY, the destination address must be the solicited node multicast address. Next, we parse the neighbor discovery options in the packet. We update the neighbor cache and the unreachability state depending on the message contents. Of course, we also send a neighbor advertisement message in response if the validity checks pass.

The neighbor advertisement message is received in response to a neighbor solicitation.

```
static void ndisc_recv_na(struct sk_buff *skb)
. . .
```

The main thing this function does is look up the advertised neighbor in the neighbor cache. This is done by calling the generic neighbor cache function, neigh_lookup. Msg points to the neighbor advertisement message contents. Target is the advertised address.

```
    neigh = neigh_lookup(&nd_tbl, &msg->target, dev);

    if (neigh) {
        if (neigh->flags & NTF_ROUTER) {
            if (msg->icmph.icmp6_router == 0) {
```

If the neighbor cache entry is listed as a router, we must change that entry back to host and try to get the actual default router.

```
                struct rt6_info *rt;
                rt = rt6_get_dflt_router(saddr, dev);
                if (rt)
                    ip6_del_rt(rt, NULL, NULL);
            }
        }  else {
```

Here we are checking the flag bit in the ICMP header of the incoming packet to see if it is coming from a router. If so, we update the neighbor cache entry.

```
            if (msg->icmph.icmp6_router)
                neigh->flags |= NTF_ROUTER;
        }
```

Here we update the neighbor cache entry by calling neigh_update. This function, discussed in Chapter 6, will update the reachability state and attach the hardware header, so packets can be transmitted using this neighbor entry.

```
        neigh_update(neigh, lladdr,
                     msg->icmph.icmp6_solicited ? NUD_REACHABLE : NUD_STALE,
                     msg->icmph.icmp6_override, 1);
        neigh_release(neigh);
    }
}
```

The Router Advertisement (RA) message serves several functions. Primarily, it is simply an RA message. When a node receives an RA message, it can determine which machine is the default or preferred router. In addition, the RA message is used for interface autoconfiguration because it may contain an address to be added to the interface's list of local addresses. The function ndisc_router_discovery processes an incoming router advertisement message.

```
static void ndisc_router_discovery(struct sk_buff *skb);
```

11.14 Multicasting and the Multicast Listener Discovery (MLD) Protocol for IPv6

In this section, we discuss the IPv6 multicast group management. Unfortunately, multicast routing is not supported by IPv6 at the time of this writing; only the host side is fully supported.

IPv4 used the Internet Group Management Protocol (IGMP) for multicast routing. In IPv4, IGMP associates a multicast address with a series of unicast addresses called a *group*. If a host wants to receive packets sent to a particular multicast destination address, it *joins* the group by sending a special IGMP report message to the all routers destination address. Routers send out IGMP queries to the all hosts group address and listen to the IGMP reports from the hosts. Then, multicast routers would forward multicast packets that have a destination address which matches the group address to all members of the group [RFC 1112].

In contrast, IPv6 is a little different. The multicast group management is now called Multicast Listener Discovery (MLD) [RFC 2710]. The old IGMP protocol used with IPv4 [RFC 1112] is not defined for IPv6. Instead, the message types are defined as ICMP messages and there is no special packet handler for ICMP. However, the initialization function for MLD does register with the socket layer, so join and drop group requests from the application can be processed. The initialization function for MLD is called igmp6_init.

```
int __init igmp6_init(struct net_proto_family *ops)
{
    struct ipv6_pinfo *np;
    struct sock *sk;
    int err;
```

When we initialize, we create a specific socket type for IGMP, igmp6_socket. Notice that the protocol is set to IPPROTO_ICMPV6 because there is no IGMP protocol defined for IPv6. Add and drop group requests are sent to us through this socket.

```
    err = sock_create(PF_INET6, SOCK_RAW, IPPROTO_ICMPV6, &igmp6_socket);
    if (err < 0) {
        printk(KERN_ERR
```

```
                "Failed to initialize the IGMP6 control socket (err %d).\ n",
                err);
        igmp6_socket = NULL; /* For safety. */
        return err;
    }

    sk = igmp6_socket->sk;
    sk->sk_allocation = GFP_ATOMIC;
    sk->sk_prot->unhash(sk);

    np = inet6_sk(sk);
    np->hop_limit = 1;
```

Here is where we create nodes for multicasting in the /proc file system.

```
#ifdef CONFIG_PROC_FS
    proc_net_fops_create("igmp6", S_IRUGO, &igmp6_mc_seq_fops);
    proc_net_fops_create("mcfilter6", S_IRUGO, &igmp6_mcf_seq_fops);
#endif

    return 0;
}
```

Instead, they are handled by ICMP as shown in Section 11.12. When ICMP receives an MLD query, it will forward it to the function igmp6_event_query.

```
int igmp6_event_query(struct sk_buff *skb);
```

This function updates the timer for the particular group, which is in the mc_gq_timer field of the inet6_dev structure. When the timer expires, the group report is sent.

When ICMP receives an MLD report, it will forward it to igmp6_event_report.

```
int igmp6_event_report(struct sk_buff *skb);
```

This function drops all reports that didn't come from a link-local source address. Next, it deletes the timer for the group. However, as of this writing, it doesn't update the routing table with the addresses in the report because IPv6 multicast routing is not supported yet.

11.15 Auto Configuration

IPv6 has two forms of address autoconfiguration. The first is called *stateless* auto-configuration, and the second is *stateful* autoconfiguration. It is important to note that IPv6 addresses are assigned to interfaces, not hosts, and unlike IPv4, interfaces can be assigned one or more addresses and can even function with no address at all. Router advertisement, part of ND discussed in Section 11.4, is the mechanism used for stateless autoconfiguration [RFC 2462]. A particular interface can self-assign a link-local address, discussed in Section 11.3. This link-local address is likely to be unique for a link-local scope, but it can be checked using duplicate address detection. This algorithm works by sending out neighbor solicitations and receiving neighbor advertisements. Another aspect of stateless autoconfiguration uses the router advertisement

message in the ND protocol [RFC 2462]. A host is informed that it is advised to use stateful autoconfiguration by a flag bit in the router discovery message.

11.16 Routing and the IPv6 FIB

The Linux IPv6 routing implementation is similar to IPv4 and shares some of the Linux networking framework. Like IPv4, route tables are used for both input and output packets because this is a flexible way of determining both machine local and external destinations in a transparent way. Like, IPv4, IPv6 contains a routing cache based on the generic destination cache. Unfortunately, though, it does not yet use the same Forwarding Information Base (FIB) as IPv4. Instead, IPv6 has its own FIB. Another difference is that the IPv4 FIB defines at least two FIB tables, a local table and a main table. IPv4 can use more FIBs if full policy routing is implemented. IPv6 has only a single FIB.

The route cache entry is defined by the rt6_info structure, defined in file *linux/include/net/ip6_fib.h*.

```
struct rt6_info
{
```

Just like IPv4, this structure is derived from a dst_entry structure.

```
    union {
        struct dst_entry      dst;
        struct rt6_info       *next;
    }  u;
```

The dev, neighbour and expires fields are in the dst_entry structure. These macros allow them to be retrieved easily from a rt6_info structure.

```
#define rt6i_dev              u.dst.dev
#define rt6i_nexthop          u.dst.neighbour
#define rt6i_expires          u.dst.expires
    struct fib6_node          *rt6i_node;
    struct in6_addr           rt6i_gateway;
    u32                       rt6i_flags;
    u32                       rt6i_metric;
    atomic_t                  rt6i_ref;
    struct rt6key             rt6i_dst;
    struct rt6key             rt6i_src;

    u8                        rt6i_protocol;
} ;
```

The search key, defined in the rt6key structure, is similar in name to the one for IPv4 in the old 2.6 kernel. However, it is much simpler.

```
struct rt6key
{
```

The key contains only an address and a prefix length. The prefix defines the network portion of the address.

```
    struct in6_addr     addr;
    int                 plen;
} ;
```

Finally, we should show how the IPv6 FIB is implemented. It is much simpler than the IPv4 FIB, which has a complex series of interleaved hash tables. The IPv6 FIB is a relatively simple tree structure. It does have the optional capability of supporting subtrees. The FIB node structure, fib6_node, is shown next. It is defined in file *linux/include/net/ip6_fib.h*.

```
struct fib6_node
{
    struct fib6_node    *parent;
    struct fib6_node    *left;
    struct fib6_node    *right;
```

Subtrees are supported if the option is configured into the kernel.

```
    struct fib6_node    *subtree;
```

The actual route is stored here. The data structure is shown above.

```
    struct rt6_info     *leaf;
```

The next field, fn_bit, is used for address prefix matching.

```
    __u16               fn_bit;
    __u16               fn_flags;
    __u32               fn_sernum;
} ;
```

The output routing function is called when we are ready to transmit a SOCK_DGRAM packet or when we are trying to establish a connection for a SOCK_STREAM type socket. In either case, the function ip6_route_output routes the output packets.

```
struct dst_entry * ip6_route_output(struct sock *sk, struct flowi *fl);
```

As is the case with IPv4, the goal of the routing function is to find and return a destination cache entry, dst_entry, which holds a route that meets the search criteria. This function is slightly different from its IPv4 counterpart because the destination cache entry is returned directly by the function. The caller updates the skb. With IPv4, the routing function returned an error code and updated the routing cache and the skb automatically.

The input routing function is called when we receive an incoming packet in IPv6 and we must determine what to do with it or who should get it next. Ip6_route_input is almost identical to the output routing function.

```
void ip6_route_input(struct sk_buff *skb);
```

This function also finds the destination cache entry in the routing table. Generally, however, the destination cache entry will point to the IPv6 input function, the IPv6 forwarding function, or the multicast input function depending on the match in the routing table.

Both of these functions call a FIB lookup function, fib6_lookup, to try to locate a route.

```
struct fib6_node * fib6_lookup(struct fib6_node *root, struct in6_addr
*daddr, struct in6_addr *saddr);
```

Fib6_lookup returns a pointer to an entry in the FIB.

11.17 Summary

In this chapter, we provided an overview of IPv6, including the address formats and header structure. We also discussed the IPv6 implementation in Linux, at least the USAGI version we used in this text. We covered how the protocol family was initialized. We covered the transport protocols, UDP and TCP for IPv6. We also discussed the routing table and neighbor cache implementation. We compared some aspects of IPv6 with IPv4. Finally, we talked about some of the member protocols in the IPv6 protocol suite, including ICMP and ND.

Appendix A: RFCs

The following is a list of all the RFCs referenced in this book. You can find them at *www.faqs.org/rfcs/* and on the companion CD-ROM.

760: DoD Standard Internet Protocol
791: Internet Protocol
792: Internet Control Message Protocol
793: Transmission Control Protocol
795: Service Mappings (Type of Service Field in IP Header)
796: Address Mappings
826: Ethernet Address Resolution Protocol: Or converting network protocol addresses to 48-bit Ethernet address for transmission on Ethernet hardware
853: Telnet Protocol Specification
896: Congestion Control in IP/TCP Internetworks (Nagle Algorithm)
917: Internet Subnets
919: Broadcasting Internet Datagrams
922: Broadcasting Internet Datagrams in the Presence of Subnets
950: Internet Standard Subnetting Procedure
1112: Host extensions for IP multicasting (IGMPv1)
1122: Requirements for Internet Hosts—Communication Layers
1256: ICMP Router Discovery Messages
1191: Path MTU Discovery
1323: TCP Extensions for High Performance
1337: TIME_WAIT Assassination Hazards in TCP
1338: Supernetting: an Address Assignment and Aggregation Strategy
1379: Extending TCP for Transactions—Concepts
1380: IESG Deliberations on Routing and Addressing
1518: An Architecture for IP Address Allocation with CIDR
1519: Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy
1583: OSPF Version 2 (Open Shortest Path First)
1661: The Point-to-Point Protocol (PPP)
1662: PPP in HDLC-like Framing
1663: PPP Reliable Transmission
1771: A Border Gateway Protocol 4 (BGP-4)
1812: Requirements for IP Version Routers
1883: Internet Protocol Version 6 (Ipv6) Specification
1884: IP Version 6 Addressing Architecture

2001: TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms

2018: TCP Selective Acknowledgement Options

2101: Ipv4 Address Behavior Today

2292: Advanced Sockets API for Ipv6

2367: PF_KEY Key Management API, Version 2

2373: IP Version 6 Addressing Architecture

2401: Security Architecture for the Internet Protocol

2402: IP Authentication Header

2406: IP Encapsulating Security Payload (ESP)

2409: The Internet Key Exchange (IKE)

2460: Internet Protocol Version 6 (Ipv6) Specification

2461: Neighbor Discovery Protocol for Ipv6

2463: Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (Ipv6) Specification

2581: TCP Congestion Control

2582: The NewReno Modification to TCP's Fast Recovery Algorithm

2625: IP and ARP over Fibre Channel

2710: Multicast Listener Discovery (MLD) for Ipv6

2861: TCP Congestion Window Validation

2914: Congestion Control Principles

3056: Connection of IPv6 Domains via IPv4 Clouds

3168: The Addition of Explicit Congestion Notification (ECN) to IP

3261: SIP: Session Initiation Protocol

3376: Internet Group Management Protocol, Version 3 (IGMP v2, v3)

3390: Increasing TCP's Initial Window

3493: Basic Socket Interface Extensions for IPv6

Appendix B: About the CD-ROM

CD Content

The companion CD-ROM contains all the sources discussed in this book, the RFCs referenced, and a copy of the GNU Public License (GPL). All sources on the CD-ROM are covered by the GPL.

Folders

linux-2.6.0-test10

The directory contains the Linux source tree including all files contained in this book. The kernel sources are patched with the USAGI patch for IPv6 that was current at the time of publication.

RFCs

This directory contains each of the RFCs referenced in this book.

usagi

This directory contains the Usagi patch dated November 11, 2003 for the 2.6.0 test10 kernel. The patch is provided both as a diff and as the compressed bz2 image.

The sources in this book and on the CD-ROM are from the 2.6.0-test10 revision of the Linux kernel with patches applied from the USAGI snapshot dated November 24, 2003. At the time of writing, the beta releases of the 2.6 kernel were very stable. However, the more recent IPv6 developments were not included in the early 2.6 kernel releases. Fortunately, USAGI provides a convenient patch file on their Web site (*http://www.linux-ipv6.org/*) for patching various Linux kernel releases; the patch is included on the companion CD-ROM. The sources in this book and on the companion CD-ROM already have had the patch applied. There are likely to be some changes between this version and the final released version of Linux 2.6.

Although the book primarily discusses source files in the networking part of the kernel, other sources from the Linux kernel are included on the CD-ROM particularly where they are used by or referred to by the IPv4 or IPv6 source code. Since there are likely to be changes between this version and the final released version of Linux 2.6, it is recommended that the reader not rely on the enclosed kernel sources for execution. It would be best to run the most recent released Linux 2.6 kernel updated with the most current USAGI stable release.

Instructions for Building the Kernel from the Sources on the CD-ROM

The sources on the CD-ROM may be used for execution but it is recommended that the reader obtain the most recent 2.6 kernel and apply the latest USAGI patches if applicable.

System Requirements

PC and Windows

- CD-ROM drive
- network interface
- 64 MB RAM
- 60 MB of disk space
- mouse or compatible pointing device
- Web browser; monitor
- Recent version of Windows such as Windows 98 or later
- Cygwin freely available from *http://cygwin.com/* is recommended. Also, a text and programming editor such as vim available from *http://www.vim.org* is re commended for browsing the source files in the book. For unzipping the bzip2 type files for Windows, a bzip2 file compression utility for Windows is available from Sourceforge at *http://gnuwin32.sourceforge.net/packages/bzip2.htm*

Linux

- For x86 architectures
- Red Hat 7.0 or later or compatible Linux distribution
- CD-ROM drive.

PPC, Macintosh, or Motorola architecture

- Yellow Dog Linux 3.0 or later
- CD-ROM drive

Sources on the CD-ROM

```
./linux-2.6.0-test10
    <kernel source tree>
usagi
    ./s20031124-2.6.0-test10
    ./s20031124-2.6.0-test10.diff
./rfcs
    rfc950.txt
    rfc922.txt
    rfc919.txt
    rfc917.txt
    rfc896.txt
    rfc853.txt
    rfc796.txt
    rfc795.txt
    rfc793.txt
    rfc792.txt
    rfc791.txt
    rfc760.txt
    rfc3493.txt
    rfc3390.txt
    rfc3261.txt
    rfc3168.txt
```

rfc3056.txt
rfc2914.txt
rfc2861.txt
rfc2710.txt
rfc2625.txt
rfc2582.txt
rfc2581.txt
rfc2463.txt
rfc2461.txt
rfc2460.txt
rfc2409.txt
rfc2406.txt
rfc2402.txt
rfc2401.txt
rfc2373.txt
rfc2367.txt
rfc2292.txt
rfc2101.txt
rfc2018.txt
rfc2001.txt
rfc1884.txt
rfc1883.txt
rfc1812.txt
rfc1771.txt
rfc1663.txt
rfc1662.txt
rfc1661.txt
rfc1583.txt
rfc1511.txt
rfc1380.txt
rfc1379.txt
rfc1338.txt
rfc1337.txt
rfc1323.txt
rfc1256.txt
rfc1191.txt
rfc1122.txt
rfc1112.txt

Bibliography

[ASCII] "Information Systems—Coded Character Sets—7-Bit American National Standard Code for Information Interchange (7-Bit ASCII)," American National Standards Institute; Document Number ANSI INCITS 4–1986 (R2002).

[BOVET02] Bovet, Daniel P., Cesati, Marco *Understanding the Linux Kernel*, 2nd Edition; December 2002; O'Reilly Sebastopol, CA; ISBN 0596002130.

[CISCOa] Cisco Systems, "Internetworking Technology Handbook," Cisco Systems, 1993–2003, Chapter "IBM System Network Architecture Protocols." Accessed March 19, 2004. *http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/ibmsna.htm*

[EDWAR00a] Edwards, Terry; "Gigahertz and Terahertz, technologies for broadband communications," 2000, Artech House, Inc, Norwood, MA; 1580530680, Chapter 1.

[ENCBRIT04a] "Telegraph." Encyclopaedia Britannica, Encyclopaedia Britannica Online. Accessed 19 March 2004; *http://search.eb.com/eb/article?eu=119000*

[ENCBRIT04b] "Teleprinter" Encyclopaedia Britannica, Encyclopaedia Britannica Online. Accessed 19 March 2004; *http://search.eb.com/eb/article?eu=73455*

[ENCBRIT04c] "Modem." Encyclopaedia Britannica, Encyclopaedia Britannica Online. Accessed 19 March 2004; *http://search.eb.com/eb/article?eu=54468*

[GALENETa] "Robert M. Metcalfe." *World of Computer Science*. 2 vols. Gale Group, 2002. Reproduced in *Biography Resource Center*. Farmington Hills, MI: The Gale Group. 2004. Available online at *http://galenet.galegroup.com/servlet/BioRC*

[GALENETb] "Jean Baptiste Joseph Fourier, Baron." *Encyclopedia of World Biography*, 2nd ed. 17 Vols. Gale Research, 1998. Reproduced in *Biography Resource Center*. Farmington Hills, MI.: The Gale Group. 2004. Available online at *http://galenet.galegroup.com/servlet/BioRC*

[GALL95] Gallmeister, Bill O., *Posix.4: Programming for the Real World*, January 1995; O'Reilly & Associates; ISBN: 1565920740.

[HAGEN02] Hagen, Silvia, *IPv6 Essentials*, July 2002, O'Reilly & Associates, Sebastopol, CA; ISBN 0-595-00125-8.

[HOUSE] House, Don Robert, "Telegraph Timeline", North American Data Communications Museum. Available online at *http://www.nadcomm.com/timeline.htm*

[HUBER02] Hubert, Bert, *Traffic Shaping for the User and Developer*. Ottawa Linux Symposium 2002. Available online at *http://ds9a.nl/ols-presentation*

[HUBHOW02] Hubert, Bert, Gregory Maxwell, Remco van Mook, Martijn van Oosterhaut, Paul B. Schroeder, Jasper Spaans, *Linux Advanced Traffic Control and Routing HOWTO*, Revision 1.1, July 7, 2002, Linux Documentation Project. Available online at *http://www.linux.org/docs/ldp/howto/Adv-Routing-HOWTO/index.html*

[IAPROT03] Internet Assigned Numbers Authority, *Protocol Numbers*. Updated January, 13, 2003. Available online at *http://www.iana.org/assignments/protocol-numbers*

[IEEE802.3] "IEEE Std 802.3–2002 Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications." IEEE Computer Society; March 8, 2002.

[ITUTV21] "300 bits per second duplex modem standardized for use in the general switched telephone network," Recommendation V.21 (11/88), Article Number E7179, International Telecommunications Union, ITU. Available online at *http://www.itu.int*

[JACOB88] Jacobson, V., "Congestion Avoidance and Control," *SIGCOMM '88*, Stanford, CA., August 1988. Available online at *http://citeseer.nj.nec.com/jacobson88congestion.html*

[JACOB93] Jacobson, V., "Re: query about TCP header on tcp-ip," September 7, 1993. Reposted by C. Partridge, "Jacobson on TCP in 30 Instructions," Usenet, comp.protocols.tcp-ip Newsgroup, Message-ID <1993Sep8.213239.28992@sics.se>, September 8, 1993. Available online at *ftp://ftp.ee.lbl.gov/email/vanj.93sep07.txt*

[KARN91] Karn, P. and C. Partridge, "Improving Round-Trip Time Estimates in Reliable Transport Protocols." *ACM Transactions on Computer Systems*, Vol. 9, No. 4, November 1991. Available online at *http://citeseer.nj.nec.com/karn01improving.html*

[KFALL96] K. Fall, S. Floyd, "Simulation-based Comparisons of Tahoe, Reno and SACK TCP," *Computer Communications Review,* ACM-SIGCOMM Vol. 26, No. 3, July 1996.

[KNUTH73] Knuth, Donald, E, *The Art of Computer Programming, Volume 1: Fundamental Algorithms, 3rd Edition*, Addison-Wesley; ISBN 0201896834.

[LISKOV90] Liskov, B., L. Shrira, and J. Wroclawski, *Efficient At-Most-Once Messages Based on Synchronized Clocks*, ACM SIGCOMM'90, Philadelphia, PA, September 1990.

[MARSH01] Marsh, Mathew G. *Policy Routing Using Linux*, March 6, 2001; SAMS, ISBN 0672320525.

[MATH96] Mathis, Matt and Jamshid Mahdavi, "Forward Acknowledgement: Refining TCP Congestion Control." Proceedings of SIGCOMM 96, August 1996.

[MATH97] Mathis, Matt and Jamshid Mahdavi, "TCP Rate-Halving with Bounding Parameters." Technical Note, FACKnotes, 1997. Available online at *http://www.psc.edu/networking/papers/FACKnotes/current/*

[MCDYSO99a] McDyson, David; Spohn, Darren; “ATM Theory and Applications,” 1999, McGraw Hill, New York, NY, ISBN 0070453462; Section 6.3.1, pp 134–137.

[MCDYSO99b] *ibid*. Appendix B, 927–955.

[MCKUS96] McKusick, Marshall Kirk, Keith Bostic, Michael J. Karels, John S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*, 1996, Addison-Wesley; ISBN 0201549794.

[NEWTON98a] Newton, Harry, *Newton’s Telecom Dictionary*, March 1998; Flatiron Publishing, New York, NY; ISBN 1-57820-023-7; pp 4, 5.

[NEWTON98b] *ibid*. pp 537, 538.

[NEWTON98c] *ibid*. pp 99.

[RBHILL] Hill, R. B. , “The Early Years of the Strowger System,” *Bell Telephone Record* (Volume XXXI No. 3, March, 1953. P. 95 et. seq.). Available online at *http://www.privateline.com/Switching/EarlyYears.html*

[RUBINI00] Alessandro Rubini, Alessandro and Corbet, Jonathan; *Linux Device Drivers, 2nd Edition*, June 2001, O’Reilly Sebastopol, CA; ISBN: 0596000081

[RUBINI00a] *ibid.* Chapter 2, pp 15–50.

[STALL93] William Stallings; *Networking Standards, A guide to OSI, ISDN, LAN and MAN Standards*, 1993, Addison-Wesley, Reading, MA; ISBN 0-201-56357-6.

[STEV94] Stevens, W. Richard, *TCP/IP Illustrated, Volume 1 The Protocols*, 1994, Addison-Wesley; ISBN: 0201633469

[STEV98] Stevens, W. Richard *UNIX Network Programming, 2nd Edition*, January 1998, Prentice Hall; ISBN: 013490012X

[STREAM93] UNIX Systems Laboratories, *STREAMS Modules and Drivers Unix SVR4.2*, June 1993, Prentice Hall, Englewood Cliffs, NJ; ISBN 0130668796

[USAGI03] Yoshifuji, Hideaki; Kazunori Miyazawa, Yuji Sekiya, Hiroshi Esaki, Jun Murai, *Linux IPv6 Networking Past, Present and Future*. Proceedings of the Linux Symposium, July 23–26, 2003, Ottawa, Canada. Available online at *http://archive.linuxsymposium.org/ols2003/Proceedings/All-Reprints/Reprint-Yoshifuji-OLS2003.pdf*

[VAHAL96] Vahalia, Uresh, *UNIX Internals The New Frontiers*, October, 1995, Prentice-Hall Inc., Upper Saddle River, NJ; ISBN 0131019082.

[WIKIPEDa] “EBCDIC,” Wikipedia, the free Web Encyclopedia. Accessed 19 March 2004; *http://en.wikipedia.org/wiki/EBCDIC*

[WIKIPEDb] “Telegraphy,” Wikipedia, the free Web Encyclopedia, Accessed 19 March 2004; *http://en.wikipedia.org/wiki/Telegraph*

[WILC03] Wilcox, Matthew, “I’ll Do It Later: Softirqs, Tasklets, Bottom Halves, Task Queues, Work Queues and Timers.” Presented at Linux Conference Australia, Perth, Australia, January 2003. Available online at *http://www.linux.org.au/conf/2003*

[X.25] “Interface between Data Terminal Equipment (DTE) and Data Circuit-terminating Equipment (DCE) for terminals operating in the packet mode and connected to public data networks by dedicated circuit,” Recommendation X.25 (10/96); International Telecommunications Union, ITU. Available online at *http://www.itu.int*

Index

A

AAL (ATM Adaption Layer), 46
ABR (Available Bit Rate), 48
Abstract Syntax Notation (ASN.1), 63
Acknowledgements
delayed acknowledgement timer, TCP, 334–336
FACK (Forward Acknowledge), 314
flow control, 26–27
SACK (Selective Acknowledgement), 310–311
Address families, 117–118, 138
Addresses
ARP, 58, 219–220
compressed form, 522
FP (Format Prefix), 523–524
gateway, 344
groups, 558
IP, 56–58
IPv6, 521–526
link-local, 521, 524
NAT (Network Address Translation), 343
unspecified, 523
Af_inet socket, 130, 159–164
Af_inet6.c, 533, 534
Af_inet6t.c, 529
Af_inet.c, 133, 159, 175, 199, 430, 439
Af_netlink.c, 168, 169
AH (Authentication Header), 230
AHDLC (Asynchronous HDLC), 33
Allocation, socket buffers, 260–261, 266–267
Ancillary data, 293
ANSI (American National Standards Institute), 17–18
Anycast address, 521
API (Application Programming Interface)
socket, system calls, 151–152
socket API, 136–145
Application layer, 11–12, 62–63
ARP (Address Resolution Protocol)
cache, 418
neighbor table, 219–220
network layer, 58
protocol initialization, 418–422
receiving and processing packets, 422–429
Arp.c, 219, 418, 419, 422, 423
ASCII protocol, 7, 29, 30

Asm-i386/socket.h, 143
Asynchronous data transmission
described, 8
vs. synchronous data transmission, 25–27
Asynchronous HDLC. *See* AHDLC (Asynchronous HDLC)
AT&T Bell Labs, 50
ATM (Asynchronous Transmission Mode)
ATM stack, 44–45
cell, 44
described, 43–44
network interfaces, 45–46
traffic classes, 47–48
UNI service class definitions, 46–48
ATM Adaption Layer. *See* AAL (ATM Adaption Layer)
ATMF (ATM Forum), 17–18
Authentication
Authentication and Authorization (AA), 24
Authentication Header (AH), 230
Authentication Header. *See* AH (Authentication Header)
AVL tree, 416–418

B

Backlog
process_backlog function, 194–196
TCP queue processing, 482–483
UDP backlog receive, 466–467
Backlog device, 194–196
Baud rate, 8
Baudot code, 6–8
BHDLC (Bit Synchronous HDLC), 33
Big endian architecture, 53
B-ISDN protocol, 44
BISYNC (binary synchronous) protocol, 7, 27–29
Bit stuffing, 30–31
Bit-oriented synchronous data transmission, 30
Bottom half, 172, 180
Bovet, Daniel, 181, 513
Broadband networking
defined, 13
See also WANs (wide area networks)
Broadcast MAC address, 53
Broadcast packets
receiving, in UDP, 462–463
who-has, 58, 221
BSD
Berkeley Software Distribution overview, 50
BSD 4.4 and mbufs, 240–241

sockets, 61
Buffers
management and TCP/IP, 65
message buffers, 237
pre-allocated fixed-sized, 237–240
TCP control buffer, 314–315
See also Socket buffers

C

C files. *See* Files
CA (Congestion Avoidance), 330
Cable modems, 13–14
Cache
ARP (Address Resolution Protocol), 418
ARP and neighbor table, 219–220
destination cache garbage collection, 211–212
destination cache, 203, 204–207
destination cache utility functions, 209–211
destination operation structure, 207–208
HH (hardware header), 203, 223–226
in_device structure, 226–230
neighbor cache, 203–204
neighbor system, 212–218
neighbor system utility functions, 220–223
overview, 202–204
rate limiting, 231–232
route, 344, 346–360
security, 230–231
sending packets using, 223–226
slab. *See* Slab cache
stackable destination, 230–231
sysctl and /proc filesystem, 231
xfrm, 230–231
Call management, 32
Capabilities functions, 147–148
Capability.h, 149
CD-ROM, included, 569
Cell, ATM, 44
Cesati, Marco, 181, 513
Chained DMA, 300
Change_mtu network interface service function, 101
Character-based synchronous data transmission, 27–29
CIDR (Classless Inter-domain Routing), 58, 520
CIR (Committed Information Rate), 42
Cloning, socket buffers, 259, 262–263
CO (Connection Oriented) service, 53–54
Collision detection, 24

Collisions and hash tables, 346
Committed Information Rate, 42
Compressed form, addresses, 522
Computer Systems Research Group (CSRG), 50
Concurrency, 65
Configuration
in_device structure, 226–230
IPv6 auto configuration, 560
stateless auto-configuration, 521, 560
TCP/IP, 231
Congestion Avoidance (CA), 330
Congestion notification fields and FR, 42–43
Connections
CO service, 53–54
connection-oriented and connectionless protocols, 12–13, 22–23
fast path, 281, 478–482
initiating, 282–289
Control characters, 27–29
Control messages, UDP, 292–294
Copying
data, minimizing, 66
data from user space to datagram, 297–298
data from user space to socket buffer, 303–306
socket buffers, 262–263
CORBA (Common Object Request Broker Architecture), 63
Count-oriented synchronous data transmission, 29–30
CSMA/CD (Carrier Sense Multiple Access with Collision Detect), 24, 53

D

Data communications
character coded transmission, 6–8
history of, 5–11
measuring speed, 8–9
methods, evolution of, 5–6
multiplexing channels, 9–11
over telephony voice networks, 9
printing telegraph, 6, 7
protocols, history of, 7
Data link layer, 11, 53–55
Datagrams
transport layer, 59, 199–202
UDP (User Datagram Protocol), 51, 59
DCE (Data Communications Equipment), 32
De-allocation, socket buffers, 260–261
Defense Advanced Research Projects Agency (DARPA), United States, 50
De-fragmentation, IPv6, 536–540
Delayed acknowledgement timer, 334–336

De-multiplexing, socket, 129
Denial-of-Service (DoS) attacks, 147–148, 417
Destination cache
described, 203, 204–207
garbage collection, 211–212
neighbor system, 212–217
sending packets using, 223
utility functions, 209–211
Destination operation structure, 207–208
Destructor function, 102
Dev.c, 91, 97, 100, 102, 104–105, 113, 172, 176, 177, 184, 187, 192
DHCP (Dynamic Host Configuration Protocol), 520–521
Dial pulses, 9
Digital data rate hierarchy in public network, 15–17
Directories
network interface drivers, 4–5
source code, 4
See also Files
Discovery, device, 86–91
DLCI (Data Link Connection Indicator), 42
DMA (Direct Memory Access), 24
Do_ioctl network interface service function, 101–102
Dp_output.c, 297, 445, 536
Driver entry points, 70
Drivers
directories for, 4–5
linux/drivers/net/e100/e100_main.c, 88, 91
linux/drivers/net/e100/e100.h, 82
linux/drivers/net/Space.c, 73
linux/drivers/pci/pci_driver.c, 89
writing, 70
See also Network interface drivers
DSL (digital subscriber line), 13–14, 25
Dst.c, 207, 209, 211, 212
Dst.h, 230
DTE (data terminal equipment), 32
Dual-Tone Multi-Frequency (DTMF), 10
Dynamic Host Configuration Protocol (DHCP), 520–521
Dynamic routing, 344

E

E100_main.c, 88, 91
E100/e100.h, 82
EBCDIC protocol, 7, 29, 30
ECN (Explicit Congestion Notification), 324
802.2 LLC (Logical Link Control), 33
Embedded systems, 63–66

Encapsulating Security Payload (ESP), 230
Encapsulation, 53
Errors, flow control and reliable transmission, 26–27
Escapes, 30
Escaping, 29
ESTABLISHED state, 493–501
Etherdevice.h, 80, 82
Ethernet chip PCI device ID, 87–88
Ethernet protocol
history of, 7
LANs, 13–14, 24–25
PPPoE (PPP over Ethernet), 40
ETSI (European Telecommunications Standards Institute), 17–18
Extension headers, 526

F

FACK (Forward Acknowledge), 314, 324
Fast path routing, 281, 386–390, 478–482
FCS (Frame Check Sequence), 24
FIB (Forward Information Base)
initialization, 370–372
interface to applications, 372–378
internal data structures, 361–364
internal kernel interface, 378–384
routes, 364–370
routing, IPv6, 560–563
RPDB, 343–345, 360–361, 521
table hash function, 371–372, 384–385
Fib_frontend.c, 360, 370, 375, 376, 379, 380, 381
Fib_hash.c, 371, 384
Fib_rules.c, 365, 367
Files
drivers, 73, 82, 88, 89, 91
folders on CD-ROM, 570–571
See also specific files
Filtering, 54
Flags
network interface, 97–98
routing control, 377–378
Flow control and reliable transmission, 26–27
Flow.h, 349
Fourier series and transforms, 10
FP (Format Prefix), 523–524
Fragmentation
IPv6 implementation, 536–540
socket buffers, 254–258
Frames

described, 14–15
Frame Relay Bearer Service (FR), 40–43
HDLC (High Level Data Link Protocol), 35–37
Frequency Division Multiplexing (FDM), 10, 15
FTP (File Transfer Protocol), 63

G

Garbage collection
destination cache, 211–212
route cache, 356–360
Gateway address, 344
Get_stats network interface service function, 101
GNU Public License, 4
Group addresses, 558
GUIs (Graphical User Interfaces), 62

H

Hard_start_xmit network interface service function, 100
Hardware architecture and TCP/IP development, 50, 63–64
Hash tables
collisions, 346
FIB (Forward Information Base), 371–372, 384–385
UDP, 463–465
HDLC (High Level Data Link Protocol)
background, 30, 33
framing, 35–37
PPP (point-to-point) protocols, 40, 54
types and configurations, 34
variations of, 33
Header files. *See* Files
Headroom and socket buffers, 254
Heap-based memory allocation, 236–237
HH (hardware header) cache, 203, 223–226
Hiding, information, 53
History of data communications, 5–11
HTTP (HyperText Transfer Protocol), 63

I

IBM Bisync protocol, 7
IBM SNA protocol, 7
ICMP (Internet Control Message Protocol)
IPv6, 552–554
overview, 429–430
packet processing, 430–435
sending packets, 435–437
Icmp.c, 429, 430
Icmp.h, 430

IDs
device discovery, 86–91
station, 12
IEC (International Electrotechnical Commission), 17–18
IEEE-SA (Institute of Electrical and Electronics Engineers Standards Association), 17–18
IETF (Internet Engineering Task Force), 17–18, 66
If_arp.h, 425
If_ether.h, 172, 173
If_inet6.h, 528
IGMP (Internet Group Management Protocol)
adding and leaving groups, 444–445
overview, 438–439
queries, 441–444
receiving packets, 439–441
reports, 444
Igmp.c, 438
Igmp.h, 439
IKE (Internet Key Exchange), 230
In6.h, 149, 525
Include files. *See* Files
Inet_peer structure, 416–418
Inet_protos array, 199–201
Inetdevice.h, 227, 228, 380
Inetpeer.h, 416
In.h, 137, 438
Init.h, 80
Initialization
ARP protocol, 418–422
device discovery, 86–91
FIB (Forward Information Base), 370–372
IP protocol, 345–346
IPv6, 529–533
neighbor system, 217–218
of net device structure, 81–86
of network devices, 80–81
packet handler, 175–179
queuing layer, 187–189
of socket layer, 128–129
sockets, in IPv4, 133–135
TCP/IP stack, 175–179
transport layer socket, 278–282
International Electrotechnical Commission (IEC), 17–18
International Standards Organization (ISO), 17–18
International Telecommunications Union (ITU), 17–18
Internet Control Message Protocol. *See* ICMP (Internet Control Message Protocol)
Internet Engineering Task Force (IETF), 17–18, 66
Internet peers, 416–418

Interrupt.h, 181, 182
IO calls and sockets, 165–166
IO control and do_ioctl function, 101–102
IP addressing, 56–58
IP layer
ARP (Address Resolution Protocol), 418–429
delivering packets to higher-level protocols, 453–456
FIB and FIB rules, 361–385
ICMP (Internet Control Message Protocol), 429–437
Internet peers, 416–418
ip_rcv input function, 447–450
ip_rcv_finish function, 450–453
IPv4 address assignment and in_device structure, 226–230
IPv4 protocol registration, 133–135
IPv4 routing, 343–345
multicast and IGMP, 438–445
protocol initialization, 345–346
receiving packets, 447–456
route cache, 346–360
routing cache, 343–345
routing input packets, 386–401
routing output packets, 401–416
routing theory, 342–343
RPDB (Routing Policy Database), 343–345, 360–361, 521
sending packets, 445–447
See also IPv6
Ip_fib.h, 361, 363, 364, 367, 375, 378, 381
Ip_output.c, 179, 189, 450
Ip_sockglue.c, 293
Ip6_fib.h, 561, 562
Ip6_input.c, 541
Ip6_output.c, 540
Ip.h, 160, 293, 353
IPOA (IP over Frame Relay), 25
IPSec, 230
Ip-sysctls.txt, 231, 232
IPv4
address assignment and in_device structure, 226–230
compared to IPv6, 520
protocol registration, 133–135
routing, 343–345
tcp_ipv4.c, 274, 279, 281, 285, 300, 472, 479
IPv6
addressing, 521–526
auto configuration, 560
compared to IPv4, 520
facilities in, 520–521

fragmentation and de-fragmentation, 536–540
ICMP, 552–554
implementation in Linux, 528–533
initialization, 529–533
input, 541–545
introduction, 520
MLD protocol, 558–560
ND (Neighbor Discovery), 554–558, 560
output, 540–541
packet format, 526–527
routing and FIB, 560–563
socket API, 149–151
socket implementation, 533–536
TCP, 548–552
UDP, 545–548
USAGI project, 520
Ipv6_sockglue.c, 532
Ipv6.h, 527
Ipv6icmp.c, 552
Ipv6upd.c, 545
IS (Intermediate System), 46
ISDN protocols, 44
ISO (International Standards Organization), 17–18
ITU (International Telecommunications Union), 17–18, 31
Iw_handler.h, 73, 75

J

Jacobson, Van, 287, 319, 320, 332, 493
Joining addresses, 558

K

Karn & Partridge algorithms, 319, 320
Keepalive timer, TCP, 336–339
Kernel
instructions for building, 569
source code, 4
tasklets, 172
threading, 179–184
Key management, 230

L

LANs (local area networks), 13–14, 23–25
LAPB (Link Access Protocol Balanced), 32, 33, 40
LAPD (Link Access Protocol D channel), 33
LAPF (Link Access Protocol for Frame Relay), 33
LAPM (Link Access Protocol Modem), 33
Latency, low, and TCP/IP, 66

Leased lines, 16
Link-local addresses, 521, 524
Linux
files. *See* Files
introduction, 3–4
TCP/IP source code, 4–5
Linux Advanced Routing & Traffic Control HOWTO (Hubert), 185
Little endian architecture, 53
Locks, socket, 164–165

M

MAC (Media Access Control)
addresses, 12, 53
header, 24
protocol example, 51
Mahdavi, Jamshid, 324
Marsh, Mathew G., 343
Mathis, Matt, 324
Maximum Transmission Unit. *See* MTU (Maximum Transmission Unit)
Mblocks, 237
Mbufs, 238–241
Memory allocation
chained DMA, 300
fragmentation and segmentation, 254–258
heap-based, 236–237
introduction, 234–235
Linux slab allocator, 242–246
mblocks, 237
mbufs, 238–241
message buffers, 237
pages, 234
pre-allocated fixed-sized buffers, 237–240
resource map allocator, 237
slab allocation, 241
socket buffers, 246–266
statistics, 266–267
Message buffers, 237
MLD (Multicast Listener Discovery), 558–560
MMU (Memory Management Unit), 151
Modems, 10, 13–14
Morse, Samuel, 5
Morse code, 5
Mrouted, 438, 444
Msghdr structure, 273
MTU (Maximum Transmission Unit)
change_mtu function, 101
described, 56

Multicasting
IGMP (Internet Group Management Protocol), 438–445
in_device structure, 226–230
IPv6 addressing, 521–526
MAC address, 53
MLD protocol for IPv6, 558–560
receiving packets in UDP, 462–463
set_multicast_list function, 99–100
Multiplexing
communication channels, 9–11
socket multiplexor, 152–153
transport layer de-multiplexing, 199–202
Multitasking, 65
Multithreading, 179–184

N

Nagle algorithm, 322
NAT (Network Address Translation), 343, 392–393
ND (Neighbor Discovery), 521, 554–558, 560
Ndisc.c, 554, 556, 557
Ndisc.h, 555
Neighbor cache, 203–204
Neighbor system
ARP and neighbor table, 219–220
described, 212–217
initialization, 217–218
utility functions, 220–223
Neighbor.c, 220
Neighbour.h, 213, 214, 220, 224
Net_device structure, 71–79
Net_init.c, 80, 82, 91
Net_rx_softirq, 191–199
Netdev_chain notifier call chain, 113
Netdevice.h, 71, 93, 99, 101, 103–104, 113, 176, 184
Netfilter, 54
Net.h, 119, 157
Netif_receive_skb function, 197–199
Netlink sockets, 146–147, 166–169
Netlink.h, 166
Netmask, 58
Network interface devices, 71
device discovery, 86–91
flags, 97–98
initialization, 80–86
network device structure, 71–79
notification, status, 110–113
receiving packets, 102–107, 191–199

registration, 91–97
service functions, 70, 97–102
transmitting packets, 107–110
Network interface drivers
directories for, 4–5
introduction, 70–71
for network interface devices, 71
Network layer, 11–12, 55–58
Networking standards, 17–18
Next hop, 344
NNI (Network-to-Network Interfaces), 44–46
Notification
congestion fields and FR, 42–43
device status, 110–112
generic event notification functions, 112–113
net_dev chain, 113
notifier chains, 110–113
Notifier.h, 111

O

OoB(Out-of-Band) data, 42, 324, 516
Open network interface service function, 98–99
OSI (Open System Interconnect) seven-layer model, 11–12, 51–53

P

Packet Handler Registration Facility, 175
Packet handlers
registration and initialization, 173–179
terminology, 172, 175
Packet sockets, 145–146
Packets
delivering input IP to higher-level protocols, 453–456
described, 14–15
in flight, 320
ICMP packet processing, 430–435
internal routing and transport layer de-multiplexing, 199–202
passing to IP output, 294–297
queuing. *See* Queues
receive-side handling, 458
receiving, 102–107, 191–199
receiving and processing ARP packets, 422–429
receiving IGMP packets, 439–441
receiving in IP, 447–456
receiving multicast and broadcast, 462–463
routing, 344
routing output, 401–405
sending from IP, 445–447

sending ICMP packets, 435–437
sending using destination cache, 223
sending using neighbor and HH cache, 223–226
transmitting, 107–110, 189–191
Pages, memory, 234
Pathnames, 4
PAWS (Protection Against Wrapped Sequence Numbers) algorithm, 323, 486
Pci_driver.c, 89
Pci.h, 88
PCM protocol, 7
Peer information, long-living, 349
Permissions and security, 147–148
Pfkeyv2.h, 230
Physical layer, 11, 53
PMTU (Path Maximum Transmission Unit), 318
Policy Routing Using Linux (Marsh), 343
Policy.h, 531
Ports, 137
POTS (Plain Old Telephone Systems), 9–10
PPP (Point-to-Point) protocols, 40, 54
PPPoE (PPP over Ethernet), 40
Prequeue processing, 478–482
Presentation layer, 11–12, 63
Printing telegraph, 6, 7
Proc filesystem, 231
Process_backlog function, 194–196
Proto structure, 270–272
Proto_ops structure, 126–127
Protocol Data Unit (PDU), 44
Protocol.c, 200, 201
Protocol.h, 131
Protocols
adding and removing from inet_protos array, 201
connection-oriented and connectionless, 12–13, 22–23
families and address families, 117–118, 138
history of, 7
internal socket functions, 155–156
sockets, 117–118
sockets and protocol switch table, 129–136
See also specific protocols
Public network, digital data rate hierarchy, 15–17
Public Switched Telephone Network (PSTN)
overview, 12, 22
X.25 protocol, 31–33
Pulse code modulation (PCM), 15
PVCs (permanent virtual circuits), 13, 23

Q

Quality of Service (QoS), 43–44, 343, 521
Queues
fast path, prequeue processing, 478–482
initialization of queuing layer, 187–189
packet queuing layer and queuing disciplines, 184–191
receiving packets and net_rx_softirq, 191–199
socket buffer, 259–260
softnet_data structure, 186–187
TCP backlog queue processing, 482–483
transmitted packets, 189–191

R

RA (Router Advertisement) message, 558
Rates
ABR (Available Bit Rate), 48
CIR (Committed Information Rate), 42
measuring speed, 8–9
for public networks, 15–17
rate limiting, 231–232
Raw sockets, 146
Reassembly.c, 538
Receiving data
receive-side handling, 458
receive-state processing, TCP, 484–493
receiving packets in UDP, 459–467
socket-level receive, UDP, 467–470
in TCP, 470–483
TCP processing data segments in established state, 493–501
TCP socket-level receive, 508–518
TIME_WAIT state, 417, 477–478, 501–508
urgent data, 516–518
Registration
network devices, 91–93, 172
packet handler, 173–179
of protocols with socket layer, 135–136
sockets, in IPv4, 133–135
utility functions, 93–97
Reports, IGMP, 444
Resource map, 237
Retransmit timer, 329–332
RFCs (Request For Comments), 3, 66–67, 565–567
Ritchie, Dennis, 237
Route cache
collisions, 346
data structures, 346–353
garbage collection, 356–360

overview, 344
Route.c, 206, 346, 353, 356, 357, 386
Route.h, 347, 353, 355, 373, 376, 381, 402
Routing
dynamic, 344
fast path, prequeue processing, 478–482
FIB (Forward Information Base), 343–345
input packets and fast path, 386–390
IPv6 FIB, 560–563
main input route resolving function, 390–401
main output route resolving function, 405–416
output packets, 401–405
packets and transport layer de-multiplexing, 199–202
RPDB (Routing Policy Database), 343–345, 360–361, 521
slow path, 386
sockets, 147
static, 344
table or cache, 344
theory, 342–343
RPDB (Routing Policy Database), 343–345, 360–361, 521
Rtnetlink and sockets, 169–170
Rtnetlink.h, 169, 170, 348, 352, 361, 375
RTOs (Retransmit Timeouts), 319
RTT (Round-Trip Time), 319, 332
RTTM (Round Trip Time Measurement), 323

S

SACK (Selective Acknowledgement), 310–311, 323–324
SAR (Segmentation and Re-assembly) firmware, 46
Scatter-gather capability, 24, 108
Sch_generic.c, 93, 108
Sched.h, 148
Scm.c, 154
SDLC (Synchronous Data Link Control), 30, 33
Security
SAs (Security Associations), 230
sockets and capabilities functions, 147–148
stack, 230–231
Segmentation and socket buffers, 254–258
Session layer, 11–12, 62–63
Set_multicast_list network interface service function, 99–100
Shift bits, 6
Signaling
described, 32
OoB(Out-of-Band) data, 42, 324, 516
Simple Network Management Protocol (SNMP), 63
SIP (Session Initiation Protocol), 63

SIT (Simple Internet Transition), 531
Sk_buff structure, 248–252
Sk_buff.h, 119
Skbuff.h, 248, 254, 256
Slab cache
introduction, 234
slab allocation, 241
utility functions, 242–246
Slab.h, 242
Sliding windows technique, 26, 38–39
Slow path routing, 386
SMP (Symmetric Multiprocessing) aware, 180
Sock structure, 119–125
Socket buffers
allocation and de-allocation, 260–261
allocation and lists, 259–260
cloning, 259, 262–263
copying, 262–263
copying data from user space to, 303–306
explanation of, 252–254
fragmentation and segmentation, 254–258
managing lists, 265–266
manipulating pointer fields, 263–265
overview, 246–247
queues, 259–260
reserving headroom, 254
shared, 259
sk_buff structure, 248–252
utility functions, 260–266
See also Queues
Socket structure, 125–126
Socket Transport Layer Interface API, 63
Socket.c, 128, 152, 154, 156, 157, 165
Socket.h, 149, 165, 273
Sockets
API, 136–145
API and IPv6, 149–151
BSD, 61
connection initiation, 282–289
corked, 162
creating, 156–159
creating af_inet socket, 130, 159–164
family values and protocol switch table, 129–136
implementation of API system calls, 151–152
initialization, 133–135, 278–282
introduction, 116
IO system calls, 165–166

layer initialization, 128–129
layer internal functions, 153–155
locks, 164–165
managing, data structures for, 119–127
msghdr structure, 273
multiplexor, 152–153
netlink, 146–147, 166–169
packet, 145–146
vs. port, 137
proto structure, 270–272
protocol internal functions, 155–156
protocols and address families, 117–118, 138
purposes of, 116–118
raw, 146
registration and initialization in IPv4, 133–135
registration of protocols with socket layer, 135–136
routing, 147
rtnetlink, 169–170
security and capabilities functions, 147–148
sending data from, via TCP, 298–308
sending data from, via UDP, 289–298
sock_dgram type, 138
sock_stream type, 138
socket layer functions, 270–278
transport layer interface, 61–62
Sock.h, 119, 162, 270, 272
SoftIRQs, 172, 180–184
Softnet_data structure, 186–187
Solaris OS, 234, 241
Sonet and SDH Optical Hierarchy, 16–17
Space.c, 73
SPD (Security Policy Database), 230
Speed of communications. *See* Rates
SSCF (Service Specific Convergence Function), 46
SSSC (Service Specific Convergence Sublayers), 46
Stacks
ATM stack, 44–45
TCP/IP, 51–63, 175–179
Standards, networking
Start transmission and hard_start_xmit function, 100
Start/stop bits, 8
Stateful autoconfiguration, 560
Stateless autoconfiguration, 521, 560
Static routing, 344
Station IDs, 12
Statistics and get_stats function, 101
Stevens, W. Richard, 62, 307, 317, 319, 321, 322

Stop and wait acknowledgement, 26
Stop network interface service function, 101
Streaming service, 59
STREAMS, 237
Strowger, Almon, 9
Strowger Switch, 9
Structure of Management Information (SMI), 63
Subnetting, 58
SVCs (switched virtual circuits), 13, 23
Switch table, protocol, 129–136
Switching
central office, 9
fabric, ATM, 46
SYN table, 325
Synchronous data transmission
vs. asynchronous data transmission, 25–27
bit stuffing, 30–31
bit-oriented, 30
character-based, 27–29
count-oriented, 29–30
Sysctl, 231
Sysctl.h, 228
System requirements, 570

T

Table, neighbor, 219–220
Taps, 175
Tasklets, 180–184
TCP (Transmission Control Protocol)
backlog queue processing, 482–483
control buffer, 314–315
copying data from user space to socket buffer, 303–306
delayed acknowledgement timer, 334–336
fast path, prequeue processing, 478–482
IPv6, 548–552
keepalive timer, 336–339
processing data segment in established state, 493–501
receive state processing, 484–493
receiving data, overview, 470–472
receiving urgent data, 516–518
retransmit timer, 329–332
sending data from sockets, 298–308
socket glue, 274
socket initialization, 278–280
socket options, 274–278
socket-level receive, 508–518
TCP_CORK, 275

TCP_DEFER_ACCEPT, 275
tcp_func structure, 281–282
TCP_INFO, 276–277
TCP_KEEPCNT, 277
TCP_KEEPIDLE, 277
TCP_KEEPINTVL, 277
TCP_LINGER2, 277
TCP_MAXSEG, 277
TCP_NODELAY, 278
tcp_opt options structure, 316–326
TCP_QUICKACK, 278
tcp_sendmsg completion, 306–308
tcp_sendmsg function, 300–303
TCP_SYNCNT, 278
tcp_timewait_state_process function, 503–508
tcp_tw_bucket structure, 502–503
tcp_v4_connect function, 285–289
tcp_v4_rcv receive handler function, 472–478
TCP_WINDOW_CLAMP, 278
TIME_WAIT state, 417, 477–478, 501–508
timers, 326–339
transmission and tcp_transmit_skb function, 309–313
window probe timer, 332–334
write timer, 327–329
Tcp_input.c, 471, 484, 493, 495
Tcp_ipv4.c, 274, 279, 281, 285, 300, 472, 479
Tcp_ipv6.c, 548
Tcp_minisocks.c, 503
Tcp_timer.c, 326, 327, 328, 332, 334, 337
Tcp.c, 300, 304, 508, 517
Tcp.h, 276, 281, 314, 316, 327, 333, 337, 479, 480, 494, 503
TCP/IP
application layer protocols, 62–63
ARP (Address Resolution Protocol), 58, 219–220
big endian architecture, 53
buffer management, 65
cache, 202–226
concurrency and multitasking, 65
configuration, 231
embedded systems, 63–66
implementation, note on, 50–51
introduction, 50
IP addressing, 56–58
kernel threading, 179–184
layer 1, physical layer, 53
layer 2, data link layer, 53–55
layer 3, network IP layer, 55–58

layer 4, transport layer, 59–61
layer 5, session layer, 62–63
layer 6, presentation, 63
layer 7, application, 63
link layer facility, 65
Linux implementation, introduction, 172–173
little endian architecture, 53
low latency, 66
minimal data copying, 66
OS requirements, 64–66
OSI model, 51–53
packet handler glue, 173–175
packet queuing layer and queuing disciplines, 184–191
segments, 61
sockets and transport layer interface, 61–62
source code, 4–5
stack initialization, 175–179
standards, numbers, and practical considerations, 66–67
tasklets, softIRQs, and timers, 180–184
TCP (Transmission Control Protocol), 60–61
timer facility, 65
UDP (User Datagram Protocol), 51, 59
Telephony voice networks, 9
Teletypes, 8
Telex protocol, 7
Text files on CD-ROM, 571–572
Threading, kernel, 179–184
Time division modulation (TDM), 15–16
Time sharing, 62
TIME_WAIT state, 417, 477–478, 501–508
 Timer.h, 183, 184
Timers
destination cache garbage collection, 211–212
kernel threading, 180–184
overview, 65
route cache garbage collection, 356–360
TCP, 326–339
Torvalds, Linux, 3
Traffic engineering and FR, 42
Transformer (XFRM), 230–231
Transmitting packets, 107–110, 189–191
Transport layer
de-multiplexing and internal packet routing, 199–202
overview, 11–12
socket initialization, 278–282
UDP and TCP, 59–62
Transport protocol dispatch process, 201–202
Trunks, 10

U

UDP (User Datagram Protocol), 51
backlog receive, 466–467
connect call, 283–285
copying data from user space to datagram, 297–298
described, 59
handling control messages, 292–294
hash table, 463–465
IPv6, 545–548
passing packet to IP output, 294–297
receiving multicast and broadcast packets, 462–463
sending data, 289–292
socket glue, 273–274
socket-level receive, 467–470
udp_rcv recieve handler function, 459–462
Udp.c, 273, 283, 290, 446, 459, 462, 463, 464, 466, 468
Udp.h, 292
Uio.h, 142
UNI (User-to-Network Interfaces), 45–48
Unicast address
IPv6, 521
MAC address, 53
Uninit network interface service function, 100
UniverSA1 Ground for IPv6 (USAGI), 4
Unix, 3, 50
Unix Internals (Vahalia), 234
Unix Network Programming (Stevens), 62
Unregistration of packet handlers, 176
Unreliable service, 59
Unspecified addresses, 523
Urgent data, 324, 516–518
USAGI project, 520
UTOPIA bus, 45

V

Van Jacobson algorithms, 319, 320, 479
VCL (Virtual Channel Links), 46
VCs (virtual circuits), 13, 22–23, 31
Visible data structure, 72
VPL (Virtual Path Links), 46

W

WANs (wide area networks), 13–14, 23–25
Who-has broadcast packet, 58, 221
Wilcox, Matthew, 180
Window probe timer, 332–334
Write timer, TCP, 327–329

X

X.25 protocol
described, 31–33
Frame Relay, 40–43
history of, 7
Xfrm.h, 230, 231
XML (Extensible Markup Language), 63

List of Figures

Chapter 2: Broadband Networking Protocols of Yesterday and Today

Figure 2.1: X.25.
Figure 2.2: HDLC unbalanced configuration.
Figure 2.3: HDLC balanced configuration.
Figure 2.4: Sliding windows.
Figure 2.5: Frame Relay header formats.
Figure 2.6: ATM stack.
Figure 2.7: ATM UNI.
Figure 2.8: ATM NNI.

Chapter 3: TCP/IP in Embedded Systems

Figure 3.1: Simple protocol example.
Figure 3.2: TCP/IP and the OSI seven-layer model.
Figure 3.3: Ethernet framing.
Figure 3.4: 802.3 framing.
Figure 3.5: IP header.
Figure 3.6: UDP header.
Figure 3.7: TCP header.
Figure 3.8: TCP segments.

Chapter 4: Linux Networking Interfaces and Device Drivers

Figure 4.1: Network interface driver registration sequence.
Figure 4.2: PCI device driver initialization sequence.

Chapter 5: Linux Sockets

Figure 5.1: AF_INET protocol family and socket calls.
Figure 5.2: Mapping of socket calls.

Chapter 6: The Linux TCP/IP Stack

Figure 6.1: Transmit packet sequence.
Figure 6.2: Receive packet sequence.
Figure 6.3: Destination cache transmit sequence.

Chapter 7: Socket Buffers and Linux Memory Allocation

Figure 7.1: Best-fit allocation here.
Figure 7.2: STREAMS mblocks.
Figure 7.3: Berkeley mbuf structure.
Figure 7.4: Linux socket buffers.
Figure 7.5: Sk_buff with fragments.
Figure 7.6: Sk_buff with segments.

Chapter 8: Sending the Data from the Socket through UDP and TCP

Figure 8.1: TCP send state diagram.
Figure 8.2: TCP transmit sequence.

Chapter 9: The Network Layer, IP

Figure 9.1: Receive packet sequence.

Chapter 10: Receiving Data in the Transport Layer, UDP, and TCP

Figure 10.1: TCP receive-side state diagram.
Figure 10.2: TCP receive packet flow.
Figure 10.3: TCP established state.

Chapter 11: Internet Protocol Version 6 (IPv6)

Figure 11.1: IPv6 header.
Figure 11.2: The fragmentation header.

List of Tables

Chapter 1: Introduction

Table 1.1: Events in the History of Data Communication
Table 1.2: The OSI Seven-Layer Model
Table 1.3: TDM Digital Hierarchy in North America
Table 1.4: TDM Digital Hierarchy in Europe
Table 1.5: Sonet and SDH Optical Hierarchy
Table 1.6: Standards Bodies

Chapter 2: Broadband Networking Protocols of Yesterday and Today

Table 2.1: BISYNC Control Characters
Table 2.2: BISYNC Frame
Table 2.3: HDLC Protocol Variants
Table 2.4: General HDLC Framing
Table 2.5: Information Format
Table 2.6: Supervisory Format
Table 2.7: Supervisory Format Commands
Table 2.8: Unnumbered Format
Table 2.9: Unnumbered Commands
Table 2.10: Frame for Frame Relay Bearer Service
Table 2.11: Explanation of Fields in Frame Relay Header
Table 2.12: ATM UNI Traffic Classes

Chapter 3: TCP/IP in Embedded Systems

Table 3.1: Protocol Types for Type Field in Ethernet MAC Header
Table 3.2: IP Addressing

Chapter 4: Linux Networking Interfaces and Device Drivers

Table 4.1: Network Queuing Layer Device State
Table 4.2: Bit Definitions for the Features Field in the Net_device Structure
Table 4.3: PCI Device ID Structure for an Ethernet Chip
Table 4.4: Network Interface Flags
Table 4.5: Network Queuing Layer Functions
Table 4.6: Queuing Layer netif_rx Return Values
Table 4.7: Pre-Declared Net Device Notification Events

Chapter 5: Linux Sockets

Table 5.1: Supported Protocol and Address Families
Table 5.2: Values for Socket Types
Table 5.3: Socket API Function Values for Flags
Table 5.4: Values for Level Argument in Setsockopt System Call
Table 5.5: Values for optname in Getsockopt and Setsockopt Calls
Table 5.6: Values for the Netlink_family Argument

Table 5.7: Inode Structure Fields Used by Sockets
Table 5.8: Socket States
Table 5.9: Values for Netlink Message Flags

Chapter 6: The Linux TCP/IP Stack

Table 6.1: Link Layer Protocol Field Values
Table 6.2: Nonofficial and Pseudo Protocol Types
Table 6.3: SoftIRQs
Table 6.4: Neighbor Cache Entry NUD States
Table 6.5: IPv4 Configuration Items

Chapter 7: Socket Buffers and Linux Memory Allocation

Table 7.1: Mbuf Structure
Table 7.2: Linux TCP/IP Slabs
Table 7.3: Slab Cache Flags
Table 7.4: Values for Flag Argument in Kmem_cache_alloc Function
Table 7.5: Packet Types in Socket Buffer pkt_type Field
Table 7.6: Active Slab Caches

Chapter 8: Sending the Data from the Socket through UDP and TCP

Table 8.1: Protocol Block Functions for UDP, Struct proto
Table 8.2: Protocol Block Functions for TCP, Struct proto
Table 8.3: IPv4 Specific Values for Tcp_func
Table 8.4: Sacked State Flags
Table 8.5: TCP Acknowledge State Values, tcp_ack_state_t
Table 8.6: Tcp_ca_state, Fast Retransmit States
Table 8.7: TCP Options
Table 8.8: TCP Transmit Timers

Chapter 9: The Network Layer, IP

Table 9.1: Route Types
Table 9.2: Route Scope Definitions
Table 9.3: Bit Definitions for ToS Field of the Flowi Structure
Table 9.4: Dst_ops values for IP Route Cache
Table 9.5: Routing Table Entry Flags
Table 9.6: Route Message Protocol Field Values
Table 9.7: IPv4 Routing Control Flags
Table 9.8: ICMP Messages

Chapter 10: Receiving Data in the Transport Layer, UDP, and TCP

Table 10.1: Receive State Flags
Table 10.2: TCP TIME_WAIT Status Values

Chapter 11: Internet Protocol Version 6 (IPv6)

Table 11.1: IPv6 Address Type Allocation
Table 11.2: Next Header Field Values

Table 11.3: Neighbor Discovery Messages

CD Content

Following are select files from this book's Companion CD-ROM. These files are for your personal use, are governed by the Books24x7 Membership Agreement, and are copyright protected by the publisher, author, and/or other third parties. Unauthorized use, reproduction, or distribution is strictly prohibited.

For more information about this content see 'About the CD-ROM'.

Click on the link(s) below to download the files to your computer:

File	Description	Size
All CD Content	The Linux TCP/IP Stack: Networking for Embedded Systems	42,599,319
Linux-2.6.0-test10		843,628
Linux-2.6.0-test10-documentation		1,815,092
Linux-2.6.0-test10-drivers		378,518
Linux-2.6.0-test10-include		2,030,442
Linux-2.6.0-test10-kernel		32,510
Linux-2.6.0-test10-net		2,512,917
Linux-2.6.0-test10-security		80,910
Rfcs		1,428,123
Usagi		33,439,001